CONTENTS

1 Getting started .......................................................... 3

2 Thanks to: ................................................................... 5

3 Table of contents .......................................................... 7
   3.1 Gallery .................................................................. 7
   3.2 Getting started ......................................................... 10
   3.3 Upgrading ............................................................... 12
   3.4 Printing (and using) formatted text ......................... 17
   3.5 Asking for input (prompts) ....................................... 21
   3.6 Dialogs .................................................................. 40
   3.7 Progress bars ........................................................ 49
   3.8 Building full screen applications ............................ 53
   3.9 Tutorials ................................................................. 59
   3.10 Advanced topics .................................................... 67
   3.11 Reference ............................................................. 88

4 Indices and tables ......................................................... 193

Python Module Index ...................................................... 195

Index ........................................................................... 197
**Warning:** Notice that this is the prompt_toolkit 3.0 documentation. It is mostly compatible with the 2.0 branch. The difference is that prompt_toolkit 3.0 requires at least Python 3.6. On the plus side, it uses asyncio natively (rather than its own event loop), and we have type annotations everywhere.

`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 starting at Python 3.6. (Python 2.6 - 3.x is supported in prompt_toolkit 2.0; not 3.0).
- Runs on Linux, OS X, OpenBSD and Windows systems.
- Lightweight, the only dependencies are Pygments 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.
Go to *getting started* and build your first prompt. Issues are tracked on the Github project.
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 2. Thanks to:
3.1 Gallery

Showcase, demonstrating the possibilities of prompt_toolkit.

3.1.1 Ptpython, a Python REPL

The prompt:

![Python REPL](image)

The configuration menu of ptpython.

![Configuration Menu](image)

The history page with its help. (This is a full-screen layout.)
This interface is meant to select multiple lines from the history and execute them together.

**Typical usage**

1. Move the `cursor up` in the history pane, until the cursor is on the first desired line.
2. Hold down the `space bar`, or press it multiple times. Each time it will select one line and move to the next one. Each selected line will appear on the right side.
3. When all the required lines are displayed on the right side, press `Enter`. This will go back to the Python REPL and show these lines as the current input. They can still be edited from there.

**Key bindings**

---

Chapter 3. Table of contents
3.1.2 Pyvim, a Vim clone

```
3.1.2 Pyvim, a Vim clone

```

---

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

3.2 Getting started

3.2.1 Installation

```bash
pip install prompt_toolkit
```

For Conda, do:

```bash
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`.

```python
from prompt_toolkit import prompt

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

3.2. Getting started 11
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 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

3.3.1 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.

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 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.

**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.

Further, 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.

**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().

Wordcompleter

WordCompleter was moved from prompt_toolkit.contrib.completers.base.WordCompleter to prompt_toolkit.completion.word_completer.WordCompleter.

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 distinguish 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:

```python
from prompt_toolkit.filters import Condition

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

### 3.3.2 Upgrading to prompt_toolkit 3.0

There are two major changes in 3.0 to be aware of:

- First, prompt_toolkit uses the asyncio event loop natively, rather then using its own implementations of event loops. This means that all coroutines are now asyncio coroutines, and all Futures are asyncio futures. Asynchronous generators became real asynchronous generators as well.
- Prompt_toolkit uses type annotations (almost) everywhere. This should not break any code, but its very helpful in many ways.

There are some minor breaking changes:

- The dialogs API had to change (see below).

#### Detecting the prompt_toolkit version

Detecting whether version 3 is being used can be done as follows:

```python
from prompt_toolkit import __version__ as ptk_version

PTK3 = ptk_version.startswith('3.0')
```

#### Fixing calls to get_event_loop

Every usage of get_event_loop has to be fixed. An easy way to do this is by changing the imports like this:

```python
if PTK3:
    from asyncio import get_event_loop
else:
    from prompt_toolkit.eventloop import get_event_loop
```
Notice that for prompt_toolkit 2.0, `get_event_loop` returns a prompt_toolkit EventLoop object. This is not an asyncio eventloop, but the API is similar.

There are some changes to the eventloop API:

<table>
<thead>
<tr>
<th>version 2.0</th>
<th>version 3.0 (asyncio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop.run_in_executor(callback)</td>
<td>loop.run_in_executor(None, callback)</td>
</tr>
<tr>
<td>loop.call_from_executor(callback)</td>
<td>loop.call_soon_threadsafe(callback)</td>
</tr>
</tbody>
</table>

**Running on top of asyncio**

For 2.0, you had tell prompt_toolkit to run on top of the asyncio event loop. Now it’s the default. So, you can simply remove the following two lines:

```python
from prompt_toolkit.eventloop.defaults import use_asyncio_event_loop
use_asyncio_event_loop()
```

There is a few little breaking changes though. The following:

```python
# For 2.0
result = await PromptSession().prompt('Say something: ', async_=True)
```

has to be changed into:

```python
# For 3.0
result = await PromptSession().prompt_async('Say something: ')
```

Further, it’s impossible to call the `prompt()` function within an asyncio application (within a coroutine), because it will try to run the event loop again. In that case, always use `prompt_async()`.

**Changes to the dialog functions**

The original way of using dialog boxes looked like this:

```python
from prompt_toolkit.shortcuts import input_dialog
result = input_dialog(title='...', text='...')
```

Now, the dialog functions return a prompt_toolkit Application object. You have to call either its `run` or `run_async` method to display the dialog. The `async_` parameter has been removed everywhere.

```python
if PTK3:
    result = input_dialog(title='...', text='...').run()
else:
    result = input_dialog(title='...', text='...')

# Or
if PTK3:
    result = await input_dialog(title='...', text='...').run_async()
else:
    result = await input_dialog(title='...', text='...', async_=True)
```
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:

```python
from prompt_toolkit import print_formatted_text

print_formatted_text('Hello world')
```

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

```python
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 four kinds of objects is called “formatted text”. There are various places in prompt toolkit, where we accept not just plain text (as a string), 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>`.

```python
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:

```python
# 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:

```python
# 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.

```python
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.

```python
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.

```python
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.

```python
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.

```python
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, '"'),
]
```

(continues on next page)
print_formatted_text(PygmentsTokens(tokens))

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

```python
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:

<table>
<thead>
<tr>
<th>pygments.Token</th>
<th>prompt_toolkit classname</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Token.Keyword</td>
<td>• &quot;class:pygments.keyword&quot;</td>
</tr>
<tr>
<td>• Token.Punctuation</td>
<td>• &quot;class:pygments.punctuation&quot;</td>
</tr>
<tr>
<td>• Token.Literal.String.Double</td>
<td>• &quot;class:pygments.literal.string.double&quot;</td>
</tr>
<tr>
<td>• Token.Text</td>
<td>• &quot;class:pygments.text&quot;</td>
</tr>
<tr>
<td>• Token</td>
<td>• &quot;class:pygments&quot;</td>
</tr>
</tbody>
</table>

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:

```python
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)
```
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.

```python
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 ask the user for input and returns the text. Just like `(raw_)input`.

```python
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.
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.

```python
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.

```python
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
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:

```python
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, style_from_pygments_cls, merge_styles
from prompt_toolkit.lexers import PygmentsLexer

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

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 this is by creating a list of style/text tuples. In the following example, we use class names to refer to the style.

```python
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 taken from the 256 color palette. If you want to have 24bit true color, this is possible by adding the color_depth=ColorDepth.TRUE_COLOR option to the prompt() function.

```python
from prompt_toolkit.output import ColorDepth
text = prompt(message, style=style, color_depth=ColorDepth.TRUE_COLOR)
```

### 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.

```python
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.

Nested completion

Sometimes you have a command line interface where the completion depends on the previous words from the input. Examples are the CLIs from routers and switches. A simple WordCompleter is not enough in that case. We want to be able to define completions at multiple hierarchical levels. NestedCompleter solves this issue:

```python
from prompt_toolkit import prompt
from prompt_toolkit.completion import NestedCompleter

completer = NestedCompleter.from_nested_dict(
    {
        'show': {
            'version': None,
            'clock': None,
            'ip': {
                'interface': {'brief'}
            }
        },
        'exit': None,
    }
)
```

(continues on next page)
Whenever there is a None value in the dictionary, it means that there is no further nested completion at that point. When all values of a dictionary would be None, it can also be replaced with a set.

**A custom completer**

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

```python
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.

```python
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:

3.5. Asking for input (prompts)
Finally, it is possible to pass formatted text for the display attribute of a Completion. This provides all the freedom you need to display the text in any possible way. It can also be combined with the style attribute. For instance:

```python
from prompt_toolkit.completion import Completer, Completion
from prompt_toolkit.formatted_text import HTML

class MyCustomCompleter(Completer):
    def get_completions(self, document, complete_event):
        yield Completion('completion1', start_position=0,
                         display=HTML('<b>completion</b><ansi:red>1</ansi:red>'),
                         style='bg:ansiyellow')
```

**Fuzzy completion**

If one possible completions is “django_migrations”, a fuzzy completer would allow you to get this by typing “djm” only, a subset of characters for this string.

Prompt_toolkit ships with a FuzzyCompleter and FuzzyWordCompleter class. These provide the means for doing this kind of “fuzzy completion”. The first one can take any completer instance and wrap it so that it becomes a fuzzy completer. The second one behaves like a WordCompleter wrapped into a FuzzyCompleter.
Complete while typing

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

```python
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.

```python
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.

```python
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 first 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 validated in real-time while the user is typing, but prompt_toolkit can also validate after the user presses the enter key:

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

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:

```python
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:

```python
from prompt_toolkit import PromptSession

session = PromptSession()

while True:
    session.prompt()
```

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

```python
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:

```python
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.

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

def bottom_toolbar():
    return HTML('<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.

```python
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.

```python
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`.

```python
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.

```python
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. "
```
```python
def print_hello():
    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.)

```python
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.

```python
from prompt_toolkit import prompt
from prompt_toolkit.application.current import get_app
from prompt_toolkit.enums import EditingMode
from prompt_toolkit.key_binding import KeyBindings

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

    # 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.

```python
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()
```

(continues on next page)
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.

```python
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.)

```python
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:

```python
from prompt_toolkit import prompt
import getpass

prompt('What is your name: ', default='{} % 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.

```python
from prompt_toolkit import prompt

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.

```python
from prompt_toolkit import prompt

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).

```python
from prompt_toolkit import prompt

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

### 3.5.14 Cursor shapes

Many terminals support displaying different types of cursor shapes. The most common are block, beam or underscore. Either blinking or not. It is possible to decide which cursor to display while asking for input, or in case of Vi input mode, have a modal prompt for which its cursor shape changes according to the input mode.

```python
from prompt_toolkit import prompt
from prompt_toolkit.cursor_shapes import CursorShape, ModalCursorShapeConfig

# Several possible values for the `cursor_shape_config` parameter:
prompt('>', cursor=CursorShape.BLOCK)
prompt('>', cursor=CursorShape.UNDERLINE)
prompt('>', cursor=CursorShape.BEAM)
```
3.5.15 Prompt in an asyncio application

**Note:** New in prompt_toolkit 3.0. (In prompt_toolkit 2.0 this was possible using a work-around).

For asyncio applications, it’s very important to never block the eventloop. However, `prompt()` is blocking, and calling this would freeze the whole application. Asyncio actually won’t even allow us to run that function within a coroutine.

The answer is to call `prompt_async()` instead of `prompt()`. The async variation returns a coroutines and is awaitable.

```python
from prompt_toolkit import PromptSession
from prompt_toolkit.patch_stdout import patch_stdout

async def my_coroutine():
    session = PromptSession()
    while True:
        with patch_stdout():
            result = await session.prompt_async('Say something: ')
            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.

3.5.16 Reading keys from stdin, one key at a time, but without a prompt

Suppose that you want to use prompt_toolkit to read the keys from stdin, one key at a time, but not render a prompt to the output, that is also possible:

```python
import asyncio

from prompt_toolkit.input import create_input
from prompt_toolkit.keys import Keys

async def main() -> None:
    done = asyncio.Event()
    input = create_input()

    def keys_ready():
        for key_press in input.read_keys():
            print(key_press)

    if key_press.key == Keys.ControlC:
        done.set()
```
with input.raw_mode():
    with input.attach(keys_ready):
        await done.wait()

if __name__ == "__main__":
    asyncio.run(main())

The above snippet will print the KeyPress object whenever a key is pressed. This is also cross platform, and should work on Windows.

### 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:

```python
from prompt_toolkit.shortcuts import message_dialog

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

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

```python
from prompt_toolkit.shortcuts import input_dialog

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

```python
from prompt_toolkit.shortcuts import yes_no_dialog

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

```python
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)
    ],
).run()
```
3.6.5 Radio list dialog

The `radiolist_dialog()` function displays a dialog with choices offered as a radio list. The values are provided as a list of tuples, each providing the return value (first element) and the displayed value (second element).

```python
from prompt_toolkit.shortcuts import radiolist_dialog

result = radiolist_dialog(
    title="RadioList dialog",
    text="Which breakfast would you like ?",
    values=[
        ("breakfast1", "Eggs and beacon"),
        ("breakfast2", "French breakfast"),
        ("breakfast3", "Equestrian breakfast")
    ]
).run()
```
3.6.6 Checkbox list dialog

The checkboxlist_dialog() has the same usage and purpose than the Radiolist dialog, but allows several values to be selected and therefore returned.

```python
from prompt_toolkit.shortcuts import checkboxlist_dialog

results_array = checkboxlist_dialog(
    title="CheckboxList dialog",
    text="What would you like in your breakfast ?",
    values=[
        ("eggs", "Eggs"),
        ("bacon", "Bacon"),
        ("croissants", "20 Croissants"),
        ("daily", "The breakfast of the day")
    ]
).run()
```

3.6.7 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.

```python
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> '),
    text='Do you want to continue?\nPress ENTER to quit.',
    style=example_style).run()
```
3.6.8 Styling reference sheet

In reality, the shortcut commands presented above build a full-screen frame by using a list of components. The two tables below allow you to get the classnames available for each shortcut, therefore you will be able to provide a custom style for every element that is displayed, using the method provided above.

Note: All the shortcuts use the Dialog component, therefore it isn’t specified explicitly below.
<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Components used</th>
</tr>
</thead>
</table>
| yes_no_dialog       | • Label  
• Button (x2)                                   |
| button_dialog       | • Label  
• Button                                           |
| input_dialog        | • TextArea  
• Button (x2)                                     |
| message_dialog      | • Label  
• Button                                           |
| radiolist_dialog    | • Label  
• RadioList  
• Button (x2)                                      |
| checkboxlist_dialog | • Label  
• CheckboxList  
• Button (x2)                                     |
| progress_dialog     | • Label  
• TextArea (locked)  
• ProgressBar                                    |
<table>
<thead>
<tr>
<th>Components</th>
<th>Available classnames</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog</td>
<td>• dialog</td>
</tr>
<tr>
<td></td>
<td>• dialog.body</td>
</tr>
<tr>
<td>Text Area</td>
<td>• text-area</td>
</tr>
<tr>
<td></td>
<td>• text-area.prompt</td>
</tr>
<tr>
<td>Label</td>
<td>• label</td>
</tr>
<tr>
<td>Button</td>
<td>• button</td>
</tr>
<tr>
<td></td>
<td>• button.focused</td>
</tr>
<tr>
<td></td>
<td>• button.arrow</td>
</tr>
<tr>
<td></td>
<td>• button.text</td>
</tr>
<tr>
<td>Frame</td>
<td>• frame</td>
</tr>
<tr>
<td></td>
<td>• frame.border</td>
</tr>
<tr>
<td></td>
<td>• frame.label</td>
</tr>
<tr>
<td>Shadow</td>
<td>• shadow</td>
</tr>
<tr>
<td>Radio List</td>
<td>• radio-list</td>
</tr>
<tr>
<td></td>
<td>• radio</td>
</tr>
<tr>
<td></td>
<td>• radio-checked</td>
</tr>
<tr>
<td></td>
<td>• radio-selected</td>
</tr>
<tr>
<td>Checkbox List</td>
<td>• checkbox-list</td>
</tr>
<tr>
<td></td>
<td>• checkbox</td>
</tr>
<tr>
<td></td>
<td>• checkbox-checked</td>
</tr>
<tr>
<td></td>
<td>• checkbox-selected</td>
</tr>
<tr>
<td>Vertical Line</td>
<td>• line</td>
</tr>
<tr>
<td></td>
<td>• vertical-line</td>
</tr>
<tr>
<td>Horizontal Line</td>
<td>• line</td>
</tr>
<tr>
<td></td>
<td>• horizontal-line</td>
</tr>
<tr>
<td>Progress Bar</td>
<td>• progress-bar</td>
</tr>
<tr>
<td></td>
<td>• progress-bar.used</td>
</tr>
</tbody>
</table>
Example

Let's customize the example of the checkboxlist_dialog.

It uses 2 Button, a CheckboxList and a Label, packed inside a Dialog. Therefore we can customize each of these elements separately, using for instance:

```python
from prompt_toolkit.shortcuts import checkboxlist_dialog
from prompt_toolkit.styles import Style

results = checkboxlist_dialog(
    title="CheckboxList dialog",
    text=": What would you like in your breakfast ?",
    values=[
        ("eggs", "Eggs"),
        ("bacon", "Bacon"),
        ("croissants", "20 Croissants"),
        ("daily", "The breakfast of the day")
    ],
    style=Style.from_dict({
        'dialog': 'bg:#d9ccbb3',
        'button': 'bg:#bfbf9a4',
        'checkbox': '#e8612c',
        'dialog.body': 'bg:#a9cfd0',
        'dialog shadow': 'bg:#c98982',
        'frame.label': '#fcaca3',
        'dialog.body label': '#fd8bb6',
    })),
).run()
```

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 for loop and when progress happens.

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

(continues on next page)
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:

```python
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.

```python
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,
```
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.

```python
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)
```

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:

```python
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(']'),
]
```

(continues on next page)
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:

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

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)
```

3.7.5 Adding key bindings and toolbar

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

```python
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 os
import time
import signal

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.
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 main 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.

$ python custom-key-bindings-tmp.py
42.6% [==========] 341/800 eta [00:04]
[f] Print "f" [x] Abort.

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:

```python
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 below.

3.8.2 I/O objects

Every Application instance requires an I/O object 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. Widgets have a `__pt_container__` function, which returns 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.

<table>
<thead>
<tr>
<th>Abstract base class</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Container</code></td>
<td><code>HSplit VSplit FloatContainer Window ScrollablePane</code></td>
</tr>
<tr>
<td><code>UIControl</code></td>
<td><code>BufferControl FormattedTextControl</code></td>
</tr>
</tbody>
</table>

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')),
)

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.

Finally, there is ScrollablePane, a container class that can be used to create long forms or nested layouts that are
scrollable as a whole.

**Focusing windows**

Focusing 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.

```python
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` and `FormattedTextControl` take a `key_bindings` argument.

### Global key bindings

Key bindings can be passed to the application as follows:

```python
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:

```python
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 `Application.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 we won’t refer to this name.

*Read more about key bindings…*
Modal containers

The following container objects take a modal argument VSplit, HSplit, and FloatContainer.

Setting modal=True makes what is called a modal container. Normally, a child container would inherit its parent key bindings. This does not apply to modal containers.

Consider a modal container (e.g. VSplit) is child of another container, its parent. Any key bindings from the parent are not taken into account if the modal container (child) 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:

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

58 Chapter 3. Table of contents
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.

```bash
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 practice.

```python
from prompt_toolkit import prompt

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

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

![Screenshot of Python command line interface]

```
jonathan# python sqlite-cli.py
> select * from mytable;
You entered: select * from mytable;
jonathan#
```

---

3.9. Tutorials 59
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, they’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.

```python
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()
```

---

```
jonathan# python sqlite-cli.py
> select * from mytable;
You entered: select * from mytable;
> select * from othertable;
You entered: select * from othertable;
>
GoodBye!
jonathan#
```
**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 Pygmentslexer to prompt_toolkit, it needs to be wrapped into a `PygmentsLexer`.

```python
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!')

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

![example SQL query output](image-url)
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.

```python
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',  
    '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.

```python
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

```

(continues on next page)

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!')

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.

```python
#!/usr/bin/env python
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', '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',
(continues on next page)```
'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]
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 More 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:

```python
from prompt_toolkit.key_binding import KeyBindings
bindings = KeyBindings()
@bindings.add('a')
def _(event):
    pass
```
"Do something if 'a' has been pressed."
...

@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 the following command in your shell, or even add it to your .bashrc.

```bash
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:
<table>
<thead>
<tr>
<th>Name</th>
<th>Possible keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escape Shift + escape</td>
<td>escape s-escape</td>
</tr>
<tr>
<td>Arrows</td>
<td>left, right, up, down</td>
</tr>
<tr>
<td>Navigation</td>
<td>home, end, delete, pageup, pagedown, insert</td>
</tr>
<tr>
<td>Control+letter</td>
<td>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</td>
</tr>
<tr>
<td>Control + number</td>
<td>c-1, c-2, c-3, c-4, c-5, c-6, c-7, c-8, c-9, c-0</td>
</tr>
<tr>
<td>Control + arrow</td>
<td>c-left, c-right, c-up, c-down</td>
</tr>
<tr>
<td>Other control keys</td>
<td>c-@, c-, c-, c-], c-^, c-, c-delete</td>
</tr>
<tr>
<td>Shift + arrow</td>
<td>s-left, s-right, s-up, s-down</td>
</tr>
<tr>
<td>Control + Shift + arrow</td>
<td>c-s-left, c-s-right, c-s-up, c-s-down</td>
</tr>
<tr>
<td>Other shift keys</td>
<td>s-delete, s-tab</td>
</tr>
<tr>
<td>F-keys</td>
<td>f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, f21, f22, f23, f24</td>
</tr>
</tbody>
</table>

There are a couple of useful aliases as well:

<table>
<thead>
<tr>
<th>Key</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>c-h</td>
<td>backspace</td>
</tr>
<tr>
<td>c-@</td>
<td>c-space</td>
</tr>
<tr>
<td>c-m</td>
<td>enter</td>
</tr>
<tr>
<td>c-i</td>
<td>tab</td>
</tr>
</tbody>
</table>

**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:

```python
@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:

```python
@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.)

```python
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.

```python
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.

```python
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.

```python
@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.

**Asyncio coroutines**

Key binding handlers can be asyncio coroutines.

```python
from prompt_toolkit.application import in_terminal

@bindings.add('x')
async def print_hello(event):
    """
    Pressing 'x' will print 5 times "hello" in the background above the prompt.
    """
    for i in range(5):
        # Print hello above the current prompt.
        async with in_terminal():
            print('hello')

        # Sleep, but allow further input editing in the meantime.
        await asyncio.sleep(1)
```

3.10. Advanced topics
If the user accepts the input on the prompt, while this coroutine is not yet finished, an `asyncio.CancelledError` exception will be thrown in this coroutine.

**Timeouts**

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

- **Application.timeoutlen**: Like Vim’s `timeoutlen` 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 `timeoutlen` 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 user 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`:

```python
@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.

**Handling SIGINT**

The SIGINT Unix signal can be handled by binding `<sigint>`. For instance:

```python
@bindings.add('<sigint>')
def _(event):
    # ...
    pass
```

This will handle a SIGINT that was sent by an external application into the process. Handling control-c should be done by binding c-c. (The terminal input is set to raw mode, which means that a c-c won’t be translated into a SIGINT.)

For a `PromptSession`, there is a default binding for `<sigint>` that corresponds to c-c: it will exit the prompt, raising a `KeyboardInterrupt` exception.
Processing `.inputrc`

GNU readline can be configured using an `.inputrc` configuration file. This file contains key bindings as well as certain settings. Right now, prompt_toolkit doesn't support `.inputrc`, 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 \texttt{style="bold underline"} where a child overrides this style partly by specifying \texttt{style="nobold bg:ansired"}.

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

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

\begin{verbatim}
# Low intensity, dark. (One or two components 0x80, the other 0x00.)
ansiblack, ansired, ansigreen, ansiyellow, ansiblue
ansimagenta, \texttt{ansi cyan}, \texttt{ansi gray}

# High intensity, bright.
ansibrightblack, ansibrightred, ansibrightgreen, ansibrightyellow
ansibrightblue, ansibrightmagenta, ansibrightcyan, ansiwhite
\end{verbatim}

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

\section*{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 \texttt{Style} can be passed as an argument to the \texttt{Application}.

\begin{verbatim}
from prompt_toolkit.layout import VSplit, Window
from prompt_toolkit.styles 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'),
])
\end{verbatim}

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 \texttt{class:} prefix twice.

\begin{verbatim}
Window(BufferControl(...), style='class:left,bottom'),
Window(BufferControl(...), style='class:left class:bottom'),
\end{verbatim}

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.
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:#aaaaa #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.

```python
from prompt_toolkit.styles 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:

```python
from prompt_toolkit.styles.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:

```python
from prompt_toolkit.styles import merge_styles

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

Color depths

There are four different levels of color depths available:

<table>
<thead>
<tr>
<th>Color Depth</th>
<th>Description</th>
<th>Value</th>
<th>Default Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>Black and white</td>
<td>ColorDepth.DEPTH_1_BIT</td>
<td>ColorDepth.MONOCROME</td>
</tr>
<tr>
<td>4 bit</td>
<td>ANSI colors</td>
<td>ColorDepth.DEPTH_4_BIT</td>
<td>ColorDepth.ANSI_COLORS_ONLY</td>
</tr>
<tr>
<td>8 bit</td>
<td>256 colors</td>
<td>ColorDepth.DEPTH_8_BIT</td>
<td>ColorDepth.DEFAULT</td>
</tr>
<tr>
<td>24 bit</td>
<td>True colors</td>
<td>ColorDepth.DEPTH_24_BIT</td>
<td>ColorDepth.TRUE_COLOR</td>
</tr>
</tbody>
</table>

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. The 24 bit true color output needs to be enabled explicitly. 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.
An application can also decide to set the color depth manually by passing a `ColorDepth` value to the `Application` object:

```python
from prompt_toolkit.output.color_depth import ColorDepth

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

### Style transformations

Prompt_toolkit supports a way to apply certain transformations to the styles near the end of the rendering pipeline. This can be used for instance to change certain colors to improve the rendering in some terminals.

One useful example is the `AdjustBrightnessStyleTransformation` class, which takes `min_brightness` and `max_brightness` as arguments which by default have 0.0 and 1.0 as values. In the following code snippet, we increase the minimum brightness to improve rendering on terminals with a dark background.

```python
from prompt_toolkit.styles import AdjustBrightnessStyleTransformation

app = Application(
    style_transformation=AdjustBrightnessStyleTransformation(
        min_brightness=0.5,  # Increase the minimum brightness.
        max_brightness=1.0,
    ),
    # ...
)
```

### 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.)

3.10. Advanced topics
The (abstract) base class is \texttt{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.

\textbf{An example}

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

\begin{verbatim}
from prompt_toolkit.application.current import get_app
from prompt_toolkit.filters import Condition

is_searching = Condition(lambda: get_app().is_searching)
\end{verbatim}

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

\begin{verbatim}
from prompt_toolkit.application.current import get_app
from prompt_toolkit.filters import Condition

@Condition
def is_searching():
    return get_app().is_searching
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

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

\begin{verbatim}
print(is_searching())
\end{verbatim}

\textbf{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.

- \texttt{has_arg}
- \texttt{has_completions}
- \texttt{has_focus}
- \texttt{buffer_has_focus}
- \texttt{has_selection}
- \texttt{has_validation_error}
- \texttt{is_aborting}
- \texttt{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:

```python
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`.

```python
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:

```python
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:

   2. 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).

   3. 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...)
4. 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

**Note:** New in prompt_toolkit 3.0. (In prompt_toolkit 2.0 this was possible using a work-around).

Prompt_toolkit 3.0 uses asyncio natively. Calling `Application.run()` will automatically run the asyncio event loop.

If however you want to run a prompt_toolkit `Application` within an asyncio environment, you have to call the `run_async` method, like this:

```python
from prompt_toolkit.application import Application

async def main():
    # Define application.
    application = Application(...)

    result = await application.run_async()
    print(result)

asyncio.get_event_loop().run_until_complete(main())
```

### 3.10.6 Unit testing

Testing user interfaces is not always obvious. Here are a few tricks for testing prompt_toolkit applications.

**PosixPipeInput and DummyOutput**

During the creation of a prompt_toolkit `Application`, we can specify what input and output device to be used. By default, these are output objects that correspond with `sys.stdin` and `sys.stdout`. In unit tests however, we want to replace these.

- For the input, we want a “pipe input”. This is an input device, in which we can programatically send some input. It can be created with `create_pipe_input()`, and that return either a `PosixPipeInput` or a `Win32PipeInput` depending on the platform.

- For the output, we want a `DummyOutput`. This is an output device that doesn’t render anything. We don’t want to render anything to `sys.stdout` in the unit tests.
Note: Typically, we don’t want to test the bytes that are written to `sys.stdout`, because these can change any time when the rendering algorithm changes, and are not so meaningful anyway. Instead, we want to test the return value from the `Application` or test how data structures (like text buffers) change over time.

So we programmatically feed some input to the input pipe, have the key bindings process the input and then test what comes out of it.

In the following example we use a `PromptSession`, but the same works for any `Application`.

```python
from prompt_toolkit.shortcuts import PromptSession
from prompt_toolkit.input import create_pipe_input
from prompt_toolkit.output import DummyOutput

def test_prompt_session():
    inp = create_pipe_input()
    try:
        inp.send_text("hello
"
)
        session = PromptSession(
            input=inp,
            output=DummyOutput(),
        )
        result = session.prompt()
    finally:
        inp.close()
    assert result == "hello"
```

In the above example, don’t forget to send the `\n` character to accept the prompt, otherwise the `Application` will wait forever for some more input to receive.

**Using an AppSession**

Sometimes it’s not convenient to pass input or output objects to the `Application`, and in some situations it’s not even possible at all. This happens when these parameters are not passed down the call stack, through all function calls.

An easy way to specify which input/output to use for all applications, is by creating an `AppSession` with this input/output and running all code in that `AppSession`. This way, we don’t need to inject it into every `Application` or `print_formatted_text()` call.

Here is an example where we use `create_app_session()`:

```python
from prompt_toolkit.application import create_app_session
from prompt_toolkit.shortcuts import print_formatted_text
from prompt_toolkit.output import DummyOutput

def test_something():
    with create_app_session(output=DummyOutput()):
        ...
        print_formatted_text("Hello world")
        ...
```
Pytest fixtures

In order to get rid of the boilerplate of creating the input, the `DummyOutput`, and the `AppSession`, we create a single fixture that does it for every test. Something like this:

```python
import pytest
from prompt_toolkit.application import create_app_session
from prompt_toolkit.input import create_pipe_input
from prompt_toolkit.output import DummyOutput

@pytest.fixture(autouse=True, scope="function")
def mock_input():
    pipe_input = create_pipe_input()
    try:
        with create_app_session(input=pipe_input, output=DummyOutput()):
            yield pipe_input
    finally:
        pipe_input.close()
```

Type checking

Prompt_toolkit 3.0 is fully type annotated. This means that if a prompt_toolkit application is typed too, it can be verified with mypy. This is complementary to unit tests, but also great for testing for correctness.

3.10.7 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 (actually asyncio) 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.

As a consequence, we will “trampoline” back and forth between two event loops.

Note: This will use a SelectorEventLoop, not the `ProactorEventLoop` (on Windows) due to the way the implementation works (contributions are welcome to make that work).

```python
from prompt_toolkit.eventloop.inputhook import set_eventloop_with_inputhook
def inputhook(inputhook_context):
    # At this point, we run the other loop. This loop is supposed to run
    # until either `inputhook_context.fileno` becomes ready for reading or
    # `inputhook_context.input_is_ready()` returns True.

    # A good way is to register this file descriptor in this other event
    # loop with a callback that stops this loop when this FD becomes ready.
    # There is no need to actually read anything from the FD.

    while True:
        ...
    set_eventloop_with_inputhook(inputhook)
```

(continues on next page)
# Any asyncio code at this point will now use this new loop, with input hook installed.

## 3.10.8 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 |      
```

(continues on next page)
implements all text manipulations and cursor movements (Like e.g. cursor_forward, insert_char or delete_word.)

<table>
<thead>
<tr>
<th>Document (text, cursor_position)</th>
</tr>
</thead>
</table>
| Accessed as the `document` property of the `Buffer` class. This is a wrapper around the text and cursor position, and contains methods for querying this data, e.g. to give the text before the cursor.

Normally after every key press, the output will be rendered again. This happens in the event loop of the `Application` where `Renderer.render` is called.
3.10.9 The rendering pipeline

This document is an attempt to describe how prompt_toolkit applications are rendered. It’s a complex but logical process that happens more or less after every key stroke. We’ll go through all the steps from the point where the user hits a key, until the character appears on the screen.

Waiting for user input

Most of the time when a prompt_toolkit application is running, it is idle. It’s sitting in the event loop, waiting for some I/O to happen. The most important kind of I/O we’re waiting for is user input. So, within the event loop, we have one file descriptor that represents the input device from where we receive key presses. The details are a little different between operating systems, but it comes down to a selector (like select or epoll) which waits for one or more file descriptor. The event loop is then responsible for calling the appropriate feedback when one of the file descriptors becomes ready.

It is like that when the user presses a key: the input device becomes ready for reading, and the appropriate callback is called. This is the read_from_input function somewhere in application.py. It will read the input from the Input object, by calling read_keys().

Reading the user input

The actual reading is also operating system dependent. For instance, on a Linux machine with a vt100 terminal, we read the input from the pseudo terminal device, by calling os.read. This however returns a sequence of bytes. There are two difficulties:

- The input could be UTF-8 encoded, and there is always the possibility that we receive only a portion of a multi-byte character.
- vt100 key presses consist of multiple characters. For instance the “left arrow” would generate something like \x1b[D. It could be that when we read this input stream, that at some point we only get the first part of such a key press, and we have to wait for the rest to arrive.

Both problems are implemented using state machines.

- The UTF-8 problem is solved using codecs.getincrementaldecoder, which is an object in which we can feed the incoming bytes, and it will only return the complete UTF-8 characters that we have so far. The rest is buffered for the next read operation.
- Vt100 parsing is solved by the Vt100Parser state machine. The state machine itself is implemented using a generator. We feed the incoming characters to the generator, and it will call the appropriate callback for key presses once they arrive. One thing here to keep in mind is that the characters for some key presses are a prefix of other key presses, like for instance, escape (\x1b) is a prefix of the left arrow key (\x1b[D). So for those, we don’t know what key is pressed until more data arrives or when the input is flushed because of a timeout.

For Windows systems, it’s a little different. Here we use Win32 syscalls for reading the console input.

Processing the key presses

The Key objects that we receive are then passed to the KeyProcessor for matching against the currently registered and active key bindings.

This is another state machine, because key bindings are linked to a sequence of key presses. We cannot call the handler until all of these key presses arrive and until we’re sure that this combination is not a prefix of another combination. For instance, sometimes people bind jj (a double j key press) to esc in Vi mode. This is convenient, but we want to make sure that pressing j once only, followed by a different key will still insert the j character as usual.
Now, there are hundreds of key bindings in prompt_toolkit (in ptpython, right now we have 585 bindings). This is mainly caused by the way that Vi key bindings are generated. In order to make this efficient, we keep a cache of handlers which match certain sequences of keys.

Of course, key bindings also have filters attached for enabling/disabling them. So, if at some point, we get a list of handlers from that cache, we still have to discard the inactive bindings. Luckily, many bindings share exactly the same filter, and we have to check every filter only once.

Read more about key bindings …

The key handlers

Once a key sequence is matched, the handler is called. This can do things like text manipulation, changing the focus or anything else.

After the handler is called, the user interface is invalidated and rendered again.

Rendering the user interface

The rendering is pretty complex for several reasons:

- We have to compute the dimensions of all user interface elements. Sometimes they are given, but sometimes this requires calculating the size of UIControl objects.
- It needs to be very efficient, because it’s something that happens on every single key stroke.
- We should output as little as possible on stdout in order to reduce latency on slow network connections and older terminals.

Calculating the total UI height

Unless the application is a full screen application, we have to know how much vertical space is going to be consumed. The total available width is given, but the vertical space is more dynamic. We do this by asking the root Container object to calculate its preferred height. If this is a VSplit or HSplit then this involves recursively querying the child objects for their preferred widths and heights and either summing it up, or taking maximum values depending on the actual layout. In the end, we get the preferred height, for which we make sure it’s at least the distance from the cursor position to the bottom of the screen.

Painting to the screen

Then we create a Screen object. This is like a canvas on which user controls can paint their content. The write_to_screen() method of the root Container is called with the screen dimensions. This will call recursively write_to_screen() methods of nested child containers, each time passing smaller dimensions while we traverse what is a tree of Container objects.

The most inner containers are Window objects, they will do the actual painting of the UIControl to the screen. This involves line wrapping the UIControl’s text and maybe scrolling the content horizontally or vertically.
Rendering to stdout

Finally, when we have painted the screen, this needs to be rendered to stdout. This is done by taking the difference of the previously rendered screen and the new one. The algorithm that we have is heavily optimized to compute this difference as quickly as possible, and call the appropriate output functions of the Output back-end. At the end, it will position the cursor in the right place.

3.11 Reference

3.11.1 Application

class prompt_toolkit.application.AppSession(input: Optional[Input] = None, output: Optional[Output] = None)

An AppSession is an interactive session, usually connected to one terminal. Within one such session, interaction with many applications can happen, one after the other.

The input/output device is not supposed to change during one session.

Warning: Always use the create_app_session function to create an instance, so that it gets activated correctly.

Parameters

- **input** – Use this as a default input for all applications running in this session, unless an input is passed to the Application explicitly.
- **output** – Use this as a default output.
class prompt_toolkit.application.Application(
    layout: Optional[prompt_toolkit.layout.layout.Layout] = None,
    style: Optional[prompt_toolkit.styles.base.BaseStyle] = None,
    include_default_pygments_style: Union[prompt_toolkit.filters.base.Filter, bool] = True,
    style_transformation: Optional[prompt_toolkit.styles.style_transformation.StyleTransformation] = None,
    key_bindings: Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase] = None,
    clipboard: Optional[prompt_toolkit.clipboard.base.Clipboard] = None,
    full_screen: bool = False,
    color_depth: Optional[Union[prompt_toolkit.output.color_depth.ColorDepth, Callable[[], Optional[prompt_toolkit.output.color_depth.ColorDepth]]]] = None,
    mouse_support: Union[prompt_toolkit.filters.base.Filter, bool] = False,
    enable_page_navigation_bindings: Optional[Union[prompt_toolkit.filters.base.Filter, bool]] = None,
    paste_mode: Union[prompt_toolkit.filters.base.Filter, bool] = False,
    editing_mode: prompt_toolkit.enums.EditingMode = EditingMode.EMACS,
    erase_when_done: bool = False,
    reverse_vi_search_direction: Union[prompt_toolkit.filters.base.Filter, bool] = False,
    min_redraw_interval: Optional[Union[float, int]] = None,
    max_render_postpone_time: Optional[Union[float, int]] = 0.01,
    refresh_interval: Optional[float] = None,
    terminal_size_polling_interval: Optional[float] = 0.5,
    cursor: Optional[Union[prompt_toolkit.cursor_shapes.CursorShape, prompt_toolkit.cursor_shapes.CursorShapeConfig]] = None,
    on_reset: Optional[ApplicationEventHandler[_AppResult]] = None,
    on_invalidate: Optional[ApplicationEventHandler[_AppResult]] = None,
    before_render: Optional[ApplicationEventHandler[_AppResult]] = None,
    after_render: Optional[ApplicationEventHandler[_AppResult]] = None,
    input: Optional[prompt_toolkit.input.base.Input] = None,
    output: Optional[prompt_toolkit.output.base.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.
- **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.

• **refresh_interval** – Automatically invalidate the UI every so many seconds. When *None* (the default), only invalidate when `invalidate` has been called.

• **terminal_size_polling_interval** – Poll the terminal size every so many seconds. Useful if the applications runs in a thread other then main thread where SIGWINCH can’t be handled, or on Windows.

**Filters:**

**Parameters**

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

• **paste_mode** – *Filter* or boolean.

• **editor_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 an *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:** (Note that the preferred way to change the input/output is by creating an *AppSession* with the required input/output objects. If you need multiple applications running at the same time, you have to create a separate *AppSession* using a `with create_app_session():` block.

**Parameters**

• **input** – *Input* instance.

• **output** – *Output* instance. (Probably Vt100_Output or Win32Output.)

**Usage:**
app = Application(...) app.run()

# Or await app.run_async()

async cancel_and_wait_for_background_tasks() → None
Cancel all background tasks, and wait for the cancellation to be done. If any of the background tasks raised an exception, this will also propagate the exception.

(If we had nurseries like Trio, this would be the __aexit__ of a nursery.)

property color_depth: prompt_toolkit.output.color_depth.ColorDepth
The active ColorDepth.

The current value is determined as follows:

• If a color depth was given explicitly to this application, use that value.

• Otherwise, fall back to the color depth that is reported by the Output implementation. If the Output class was created using output.defaults.create_output, then this value is coming from the $PROMPT_TOOLKIT_COLOR_DEPTH environment variable.

cpr_not_supported_callback() → None
Called when we don’t receive the cursor position response in time.

create_background_task(coroutine: Awaitable[None]) → asyncio.Task[None]
Start a background task (coroutine) for the running application. When the Application terminates, unfinished background tasks will be cancelled.

If asyncio had nurseries like Trio, we would create a nursery in Application.run_async, and run the given coroutine in that nursery.

Not threadsafe.

property current_buffer: prompt_toolkit.buffer.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.)

property current_search_state: prompt_toolkit.search.SearchState
Return the current SearchState. (The one for the focused BufferControl.)

exit() → None
exit(*, result: prompt_toolkit.application.application._AppResult, style: str = "") → None
exit(*, exception: Union[BaseException, Type[BaseException]], style: str = "") → None
Exit application.

Note: If Application.exit is called before Application.run() is called, then the Application won’t exit (because the Application.future doesn’t correspond to the current run). Use a pre_run hook and an event to synchronize the closing if there’s a chance this can happen.

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() → List[str]
Return a list of used style strings. This is helpful for debugging, and for writing a new Style.

invalidate() → None
Thread safe way of sending a repaint trigger to the input event loop.

property invalidated: bool
True when a redraw operation has been scheduled.

property is_running: bool
True when the application is currently active/running.

key_processor
The InputProcessor instance.

print_text(text: Optional[Union[str, MagicFormattedText, List[Union[str, str]], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]], Callable[[], Any]], style: Optional[prompt_toolkit.styles.base.BaseStyle] = None) → 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.

quoted_insert
Quoted insert. This flag is set if we go into quoted insert mode.

render_counter
Render counter. This one is increased every time the UI is rendered. It can be used as a key for caching certain information during one rendering.

reset() → None
Reset everything, for reading the next input.

run(pre_run: Optional[Callable[[], None]] = None, set_exception_handler: bool = True, handle_sigint: bool = True, in_thread: bool = False) → prompt_toolkit.application.application._AppResult
A blocking ‘run’ call that waits until the UI is finished.

This will start the current asyncio event loop. If no loop is set for the current thread, then it will create a new loop. If a new loop was created, this won’t close the new loop (if in_thread=False).

Parameters
• pre_run – Optional callable, which is called right after the “reset” of the application.
• 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.
• in_thread – When true, run the application in a background thread, and block the current thread until the application terminates. This is useful if we need to be sure the application won’t use the current event loop (asyncio does not support nested event loops). A new event loop will be created in this background thread, and that loop will also be closed when the background thread terminates. When this is used, it’s especially important to make sure that all asyncio background tasks are managed through get_app().create_background_task(), so that unfinished tasks are properly cancelled before the event loop is closed. This is used for instance in ptpython.
• handle_sigint – Handle SIGINT signal. Call the key binding for Keys.SIGINT. (This only works in the main thread.)
```
async run_async(pre_run: Optional[Callable[[], None]] = None, set_exception_handler: bool = True, handle_sigint: bool = True, slow_callback_duration: float = 0.5) →
prompt_toolkit.application.application._AppResult

Run the prompt_toolkit Application until exit() has been called. Return the value that was passed to exit().

This is the main entry point for a prompt_toolkit Application and usually the only place where the event loop is actually running.

Parameters

- **pre_run** – Optional callable, which is called right after the “reset” of the application.
- **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.
- **handle_sigint** – Handle SIGINT signal if possible. This will call the <sigint> key binding when a SIGINT is received. (This only works in the main thread.)
- **slow_callback_duration** – Display warnings if code scheduled in the asyncio event loop takes more time than this. The asyncio default of 0.1 is sometimes not sufficient on a slow system, because exceptionally, the drawing of the app, which happens in the event loop, can take a bit longer from time to time.

async run_system_command(command: str, wait_for_enter: bool = True, display_before_text: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]], Callable[[], Any]] = ”, wait_text: str = ’Press ENTER to continue...’) →
None

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: bool = True) → None

(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.)

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.

ttimeoutlen

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 timeoutlen option in Vim.

vi_state

Vi state. (For Vi key bindings.)
class prompt_toolkit.application.DummyApplication

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

run(pre_run: Optional[Callable[[], None]] = None, set_exception_handler: bool = True, handle_sigint: bool = True, in_thread: bool = False) → None

A blocking ‘run’ call that waits until the UI is finished.

This will start the current asyncio event loop. If no loop is set for the current thread, then it will create a new loop. If a new loop was created, this won’t close the new loop (if in_thread=False).

Parameters

• pre_run – Optional callable, which is called right after the “reset” of the application.

• 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.

• in_thread – When true, run the application in a background thread, and block the current thread until the application terminates. This is useful if we need to be sure the application won’t use the current event loop (asyncio does not support nested event loops). A new event loop will be created in this background thread, and that loop will also be closed when the background thread terminates. When this is used, it’s especially important to make sure that all asyncio background tasks are managed through get_app().create_background_task(), so that unfinished tasks are properly cancelled before the event loop is closed. This is used for instance in ptpython.

• handle_sigint – Handle SIGINT signal. Call the key binding for Keys.SIGINT. (This only works in the main thread.)

async run_async(pre_run: Optional[Callable[[], None]] = None, set_exception_handler: bool = True, handle_sigint: bool = True, slow_callback_duration: float = 0.5) → None

Run the prompt_toolkit Application until exit() has been called. Return the value that was passed to exit().

This is the main entry point for a prompt_toolkit Application and usually the only place where the event loop is actually running.

Parameters

• pre_run – Optional callable, which is called right after the “reset” of the application.

• 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.

• handle_sigint – Handle SIGINT signal if possible. This will call the <sigint> key binding when a SIGINT is received. (This only works in the main thread.)

• slow_callback_duration – Display warnings if code scheduled in the asyncio event loop takes more time than this. The asyncio default of 0.1 is sometimes not sufficient on a slow system, because exceptionally, the drawing of the app, which happens in the event loop, can take a bit longer from time to time.

async run_system_command(command: str, wait_for_enter: bool = True, display_before_text: Optional[Union[Callable[[str], None], MagicFormattedText, List[Union[ Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]]], Callable[[], Any]] = ", wait_text: str = ") → None

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: bool = True) → None*

(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.)

```python
```

Create a separate AppSession.

This is useful if there can be multiple individual `AppSession`s going on. Like in the case of an Telnet/SSH server. This functionality uses contextvars and requires at least Python 3.7.

```python
prompt_toolkit.application.get_app() → Application[Any]
```

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.)

```python
prompt_toolkit.application.get_app_or_none() → Optional[Application[Any]]
```

Get the current active (running) Application, or return `None` if no application is running.

```python
prompt_toolkit.application.in_terminal(render_cli_done: bool = False) → AsyncGenerator[None, None]
```

Asynchronous context manager that suspends the current application and runs the body in the terminal.

```python
async def f():
    async with in_terminal():
        call_some_function()
        await call_some_async_function()
```

```python
prompt_toolkit.application.run_in_terminal(func: Callable[[], _T], render_cli_done: bool = False, in_executor: bool = False) → Awaitable[_T]
```

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.set_app(app: Application[Any])` ➔ `Generator[None, None, None]`

Context manager that sets the given *Application* active in an *AppSession*.

This should only be called by the *Application* itself. The application will automatically be active while its running. If you want the application to be active in other threads/coroutines, where that’s not the case, use `contextvars.copy_context()`, or use `Application.context` to run it in the appropriate context.

### 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.

**class** `prompt_toolkit.formatted_text.ANSI(value: str)`

*ANSI* formatted text. Take something ANSI escaped text, for use as a formatted string. E.g.

```python
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.

`format(*args: str, **kwargs: str)` ➔ `prompt_toolkit.formatted_text.ansi.ANSI`

Like `str.format`, but make sure that the arguments are properly escaped. (No ANSI escapes can be injected.)

**class** `prompt_toolkit.formatted_text.FormattedText(iterable=(), /)`

A list of *(style, text)* tuples.

(In some situations, this can also be *(style, text, mouse_handler)* tuples.)

**class** `prompt_toolkit.formatted_text.HTML(value: str)`

*HTML* formatted text. Take something HTML-like, for use as a formatted string. Take something HTML-like, for use as a formatted string.

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

# Italic, bold, underline and strike.
HTML(‘<i>...</i>’)  
HTML(‘<b>...</b>’)  
HTML(‘<u>...</u>’)  
HTML(‘<s>...</s>’)  
```

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: object, **kwargs: object)` ➔ `prompt_toolkit.formatted_text.html.HTML`

Like `str.format`, but make sure that the arguments are properly escaped.
class prompt_toolkit.formatted_text.PygmentsTokens(token_list: List[ Tuple[Token, str] ])
    Turn a pygments token list into a list of prompt_toolkit text fragments ( (style_str, text) tuples).

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

Example:

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

    Parameters  text – Plain text.

prompt_toolkit.formatted_text.fragment_list_len(fragments: List[ Union[ Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]] ] ]) → int

    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_to_text(fragments: List[ Union[ Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]] ] ]) → str

    Concatenate all the text parts again.

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

prompt_toolkit.formatted_text.fragment_list_width(fragments: List[ Union[ Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]] ] ]) → int

    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.is_formatted_text(value: object) → TypeGuard[ AnyFormattedText ]

    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.


    Merge (Concatenate) several pieces of formatted text together.

prompt_toolkit.formatted_text.split_lines(fragments: List[ Union[ Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]] ] ]) → Iterable[ List[ Union[ Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]] ] ] ]

    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.

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

**prompt_toolkit.formatted_text.to_formatted_text**

```python
prompt_toolkit.formatted_text.to_formatted_text(
    value: Optional[Union[str, MagicFormattedText,
                         List[Union[Tuple[str, str], Tuple[str, str,
                                    Callable[[prompt_toolkit.mouse_events.MouseEvent],
                                    None]]], Callable[[], Any]]],
    style: str = '',
    auto_convert: bool = False) \rightarrow FormattedText
```

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 always a `FormattedText` instance, which is a list of (style, text) tuples.

It can take a plain text string, an HTML or ANSI object, anything that implements `__pt_formatted_text__` or a callable that takes no arguments and returns one of those.

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.to_plain_text**

```python
prompt_toolkit.formatted_text.to_plain_text(
    value: Optional[Union[str, MagicFormattedText,
                         List[Union[Tuple[str, str], Tuple[str, str,
                                    Callable[[prompt_toolkit.mouse_events.MouseEvent],
                                    None]]], Callable[[], Any]]],
    ) \rightarrow str
```

Turn any kind of formatted text back into plain text.

### 3.11.3 Buffer

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

```python
class prompt_toolkit.buffer.Buffer(completer: Optional[prompt_toolkit.completion.base.Completer] = None,
                                    auto_suggest: Optional[prompt_toolkit.auto_suggest.AutoSuggest] = None,
                                    history: Optional[prompt_toolkit.history.History] = None,
                                    validator: Optional[prompt_toolkit.validation.Validator] = None,
                                    tempfile_suffix: Union[str, Callable[[], str]] = '',
                                    tempfile: Union[str, Callable[[], str]] = '',
                                    name: str = '',
                                    complete_while_typing: Optional[prompt_toolkit.filters.base.Filter] = False,
                                    validate_while_typing: Optional[prompt_toolkit.filters.base.Filter] = False,
                                    enable_history_search: Optional[prompt_toolkit.filters.base.Filter] = False,
                                    document: Optional[prompt_toolkit.document.Document] = None,
                                    accept_handler: Optional[Callable[[prompt_toolkit.buffer.Buffer], None]] = None,
                                    read_only: Optional[prompt_toolkit.filters.base.Filter] = False,
                                    multiline: Optional[prompt_toolkit.filters.base.Filter] = True,
                                    on_text_changed: Optional[Callable[[prompt_toolkit.buffer.Buffer], None]] = None,
                                    on_text_insert: Optional[Callable[[prompt_toolkit.buffer.Buffer], None]] = None,
                                    on_cursor_position_changed: Optional[Callable[[prompt_toolkit.buffer.Buffer], None]] = None,
                                    on_completions_changed: Optional[Callable[[prompt_toolkit.buffer.Buffer], None]] = None,
                                    on_suggestion_set: Optional[Callable[[prompt_toolkit.buffer.Buffer], None]] = 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
• **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.

• **tempfile** – For more advanced tempfile situations where you need control over the subdirectories and filename. For a Git Commit Message, this would be “.git/COMMIT_EDITMSG”, 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** – Called when the buffer input is accepted. (Usually when the user presses enter.) The accept handler receives this *Buffer* as input and should return True when the buffer text should be kept instead of calling reset.

In case of a *PromptSession* for instance, we want to keep the text, because we will exit the application, and only reset it during the next run.

**Events:**

**Parameters**

• **on_text_changed** – When the buffer text changes. (Callable or None.)

• **on_text_insert** – When new text is inserted. (Callable or None.)

• **on_cursor_position_changed** – When the cursor moves. (Callable or None.)

• **on_completions_changed** – When the completions were changed. (Callable or None.)

• **on_suggestion_set** – When an auto-suggestion text has been set. (Callable or 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**() → None

Append the current input to the history.

**apply_completion**(completion: prompt_toolkit.completion.base.Completion) → None

Insert a given completion.
apply_search(search_state: prompt_toolkit.search.SearchState, include_current_position: bool = True, count: int = 1) → None
Apply search. If something is found, set working_index and cursor_position.

auto_down(count: int = 1, go_to_start_of_line_if_history_changes: bool = False) → None
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: int = 1, go_to_start_of_line_if_history_changes: bool = False) → None
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() → None
Cancel completion, go back to the original text.

copy_selection(_cut: bool = False) → prompt_toolkit.clipboard.base.ClipboardData
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:

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

cursor_down(count: int = 1) → None
(for multiline edit). Move cursor to the next line.

cursor_up(count: int = 1) → None
(for multiline edit). Move cursor to the previous line.

cut_selection() → prompt_toolkit.clipboard.base.ClipboardData
Delete selected text and return ClipboardData instance.

delete(count: int = 1) → str
Delete specified number of characters and return the deleted text.

delete_before_cursor(count: int = 1) → str
Delete specified number of characters before cursor and return the deleted text.

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

generate_search_position(search_state: prompt_toolkit.search.SearchState, include_current_position: bool = True, count: int = 1) → int
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: Optional[int]) → None
Select a completion from the list of current completions.

go_to_history(index: int) → None
Go to this item in the history.
**history_backward** *(count: int = 1) → None*
Move backwards through history.

**history_forward** *(count: int = 1) → None*
Move forwards through the history.

**Parameters**

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

**insert_line_above** *(copy_margin: bool = True) → None*
Insert a new line above the current one.

**insert_line_below** *(copy_margin: bool = True) → None*
Insert a new line below the current one.

**insert_text** *(data: str, overwrite: bool = False, move_cursor: bool = True, fire_event: bool = True) → None*
Insert characters at cursor position.

**Parameters**

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

**property is_returnable** : bool
True when there is something handling accept.

**join_next_line** *(separator: str = '') → None*
Join the next line to the current one by deleting the line ending after the current line.

**join_selected_lines** *(separator: str = '') → None*
Join the selected lines.

**load_history_if_not_yet_loaded** () → None
Create task for populating the buffer history (if not yet done).

**Note:**

This needs to be called from within the event loop of the application, because history loading is async, and we need to be sure the right event loop is active. Therefore, we call this method in the `BufferControl.create_content`.

There are situations where prompt_toolkit applications are created in one thread, but will later run in a different thread (Ptython is one example. The REPL runs in a separate thread, in order to prevent interfering with a potential different event loop in the main thread. The REPL UI however is still created in the main thread.) We could decide to not support creating prompt_toolkit objects in one thread and running the application in a different thread, but history loading is the only place where it matters, and this solves it.

**newline** *(copy_margin: bool = True) → None*
Insert a line ending at the current position.

**open_in_editor** *(validate_and_handle: bool = False) → asyncio.Task[None]*
Open code in editor.

This returns a future, and runs in a thread executor.
paste_clipboard_data(data: prompt_toolkit.clipboard.base.ClipboardData, paste_mode:
prompt_toolkit.selection.PasteMode = PasteMode.EMACS, count: int = 1) → None
Insert the data from the clipboard.

→ None

Parameters append_to_history – Append current input to history first.

save_to_undo_stack(clear_redo_stack: bool = True) → None
Safe current state (input text and cursor position), so that we can restore it by calling undo.

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: bool = False, select_last: bool = False, insert_common_part: bool = False,
complete_event: Optional[prompt_toolkit.completion.base.CompleteEvent] = None) → None
Start asynchronous autocompletion of this buffer. (This will do nothing if a previous completion was still in progress.)

start_history_lines_completion() → None
Start a completion based on all the other lines in the document and the history.

start_selection(selection_type: prompt_toolkit.selection.SelectionType = SelectionType.CHARACTERS)
→ None
Take the current cursor position as the start of this selection.

swap_characters_before_cursor() → None
Swap the last two characters before the cursor.

transform_current_line(transform_callback: Callable[[str], str]) → None
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: Iterable[int], transform_callback: Callable[[str], str]) → str
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:

```python
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_: int, to: int, transform_callback: Callable[[str], str]) → None*

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: bool = False) → bool*

Returns True if valid.

Parameters

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

**validate_and_handle** () → None

Validate buffer and handle the accept action.

**yank_last_arg** *(n: Optional[int] = None) → None*

Like yank_nth_arg, but if no argument has been given, yank the last word by default.

**yank_nth_arg** *(n: Optional[int] = None, _yank_last_arg: bool = False) → None*

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.

completions: Optional[List[prompt_toolkit.completion.base.Completion]] = None, complete_index: Optional[int] = None)*

Immutable class that contains a completion state.

**complete_index**

Position in the completions array. This can be None to indicate “no completion”, the original text.

**completions**

List of all the current Completion instances which are possible at this point.

**property current_completion: Optional[prompt_toolkit.completion.base.Completion]**

Return the current completion, or return None when no completion is selected.

**go_to_index** *(index: Optional[int]) → None*

Create a new CompletionState object with the new index.

When index is None deselect the completion.

**new_text_and_position** () → Tuple[str, int]

Return (new_text, new_cursor_position) for this completion.

**original_document**

Document as it was when the completion started.

**exception** prompt_toolkit.buffer>EditReadOnlyBuffer

Attempt editing of read-only Buffer.

3.11. Reference
prompt_toolkit.buffer.indent(buffer: prompt_toolkit.buffer.Buffer, from_row: int, to_row: int, count: int = 1) → None

Indent text of a Buffer object.

prompt_toolkit.buffer.reshape_text(buffer: prompt_toolkit.buffer.Buffer, from_row: int, to_row: int) → None

Reformat text, taking the width into account. to_row is included. (Vi ‘gq’ operator.)

prompt_toolkit.buffer.unindent(buffer: prompt_toolkit.buffer.Buffer, from_row: int, to_row: int, count: int = 1) → None

Unindent text of a Buffer object.

### 3.11.4 Selection

Data structures for the selection.

```python
class prompt_toolkit.selection.PasteMode(value)

An enumeration.
```

```python
class prompt_toolkit.selection.SelectionState(original_cursor_position: int = 0, type: prompt_toolkit.selection.SelectionType = SelectionType.CHARACTERS)

State of the current selection.
```

**Parameters**

- original_cursor_position – int
- type – SelectionType

```python
class prompt_toolkit.selection.SelectionType(value)

Type of selection.
```

- BLOCK = 'BLOCK'
  A block selection. (Visual-Block in Vi.)

- CHARACTERS = 'CHARACTERS'
  Characters. (Visual in Vi.)

- LINES = 'LINES'
  Whole lines. (Visual-Line in Vi.)

### 3.11.5 Clipboard

```python
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.)
```

```python
abstract get_data() → prompt_toolkit.clipboard.base.ClipboardData

Return clipboard data.
```

```python
rotate() → None

For Emacs mode, rotate the kill ring.
```

```python
abstract set_data(data: prompt_toolkit.clipboard.base.ClipboardData) → None

Set data to the clipboard.
```

**Parameters**

- data – ClipboardData instance.
**set_text** *(text: str) → None*
Shortcut for setting plain text on clipboard.

**class promtoolkit.clipboard.ClipboardData** *(text: str = ", type: promtoolkit.selection.SelectionType = SelectionType.CHARACTERS)*

Text on the clipboard.

**Parameters**

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

**class promtoolkit.clipboard.DummyClipboard**
Clipboard implementation that doesn’t remember anything.

**get_data** () → promtoolkit.clipboard.base.ClipboardData
Return clipboard data.

**rotate** () → None
For Emacs mode, rotate the kill ring.

**set_data** *(data: promtoolkit.clipboard.base.ClipboardData) → None*
Set data to the clipboard.

**Parameters**

- **data** – ClipboardData instance.

**set_text** *(text: str) → None*
Shortcut for setting plain text on clipboard.

**class promtoolkit.clipboard.DynamicClipboard** *(get_clipboard: Callable[[], Optional[promtoolkit.clipboard.base.Clipboard]])*

Clipboard class that can dynamically returns any Clipboard.

**Parameters**

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

**get_data** () → promtoolkit.clipboard.base.ClipboardData
Return clipboard data.

**rotate** () → None
For Emacs mode, rotate the kill ring.

**set_data** *(data: promtoolkit.clipboard.base.ClipboardData) → None*
Set data to the clipboard.

**Parameters**

- **data** – ClipboardData instance.

**set_text** *(text: str) → None*
Shortcut for setting plain text on clipboard.

**class promtoolkit.clipboard.InMemoryClipboard** *(data: Optional[promtoolkit.clipboard.base.ClipboardData] = None, max_size: int = 60)*

Default clipboard implementation. Just keep the data in memory.

This implements a kill-ring, for Emacs mode.

**get_data** () → promtoolkit.clipboard.base.ClipboardData
Return clipboard data.

**rotate** () → None
For Emacs mode, rotate the kill ring.

**set_data** *(data: promtoolkit.clipboard.base.ClipboardData) → None*
Set data to the clipboard.
Parameters \(\text{data} \rightarrow \text{ClipboardData} \text{ instance.}\)

```python
class prompt_toolkit.clipboard.pyperclip.PyperclipClipBoard
ClipBoard that synchronizes with the Windows/Mac/Linux system clipboard, using the pyperclip module.

def get_data() \rightarrow prompt_toolkit.clipboard.base.ClipboardData
Return clipboard data.

def set_data(data: prompt_toolkit.clipboard.base.ClipboardData) \rightarrow None
Set data to the clipboard.
```

Parameters \(\text{data} \rightarrow \text{ClipboardData} \text{ instance.}\)

### 3.11.6 Auto completion

```python
class prompt_toolkit.completion.CompleteEvent
(text_inserted: bool = False, completion_requested: bool = False)
```

Event that called the completer.

Parameters

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

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

- \(\text{completion_requested} \)
  - Used explicitly requested completion by pressing ‘tab’.
- \(\text{text_inserted} \)
  - Automatic completion while typing.

```python
class prompt_toolkit.completion.Completer
Base class for completer implementations.

@abstractmethod
This should be a generator that yields \text{Completion} \text{ instances.}
If the generation of completions is something expensive (that takes a lot of time), consider wrapping this \text{Completer} class in a \text{ThreadedCompleter}. In that case, the completer algorithm runs in a background thread and completions will be displayed as soon as they arrive.

Parameters

- \(\text{document} \) – \text{Document} \text{ instance.}
- \(\text{complete_event} \) – \text{CompleteEvent} \text{ instance.}

Asynchronous generator for completions. \((\text{Probably, you won’t have to override this.})\)
Asynchronous generator of \text{Completion} \text{ objects.}
```
class prompt_toolkit.completion.Completion(text: str, start_position: int = 0, display: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]]] = None, display_meta: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]]] = None, style: str = ts1, selected_style: str = ts1)

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 or formatted text) If the completion has to be displayed differently in the completion menu.

• display_meta – (Optional string or formatted text) 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.

property display_meta: List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]
Return meta-text. (This is lazy when using a callable).

property display_meta_text: str
The ‘meta’ field as plain text.

property display_text: str
The ‘display’ field as plain text.

new_completion_from_position(position: int) → prompt_toolkit.completion.base.Completion
(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.


Wrapper around any other completer that will enable/disable the completions depending on whether the received condition is satisfied.

Parameters

• completer – Completer instance.

• filter – Filter instance.

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.

```python
AsyncGenerator[prompt_toolkit.completion.base.Completion, None]
```

Asynchronous generator for completions. (Probably, you won’t have to override this.)

Asynchronous generator of `Completion` objects.

```python
class prompt_toolkit.completion.DeduplicateCompleter(completer: prompt_toolkit.completion.base.Completer)
```

Wrapper around a completer that removes duplicates. Only the first unique completions are kept.

Completions are considered to be a duplicate if they result in the same document text when they would be applied.

```python
Iterable[prompt_toolkit.completion.base.Completion]
```

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.

```python
class prompt_toolkit.completion.DummyCompleter
```

A completer that doesn’t return any completion.

```python
Iterable[prompt_toolkit.completion.base.Completion]
```

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.

```python
class prompt_toolkit.completion.DynamicCompleter(get_completer: Callable[[], Optional[prompt_toolkit.completion.base.Completer]])
```

Completer class that can dynamically returns any Completer.

**Parameters**

- **get_completer** – Callable that returns a `Completer` instance.
get_completions(
document: prompt_toolkit.document.Document,
complete_event:
prompt_toolkit.completion.base.CompleteEvent) ➞ 
Iterable[prompt_toolkit.completion.base.Completion]

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

- complete_event – CompleteEvent instance.

get_completions_async(
document: prompt_toolkit.document.Document,
complete_event:
prompt_toolkit.completion.base.CompleteEvent) ➞ 
AsyncGenerator[prompt_toolkit.completion.base.Completion, None]

Asynchronous generator for completions. (Probably, you won’t have to override this.)

Asynchronous generator of Completion objects.

class prompt_toolkit.completion.ExecutableCompleter

Complete only executable files in the current path.

class prompt_toolkit.completion.FuzzyCompleter(
completer: prompt_toolkit.completion.base.Completer,
WORD: bool = False, pattern: Optional[str] = None,

Fuzzy completion. This wraps any other completer and turns it into a fuzzy completer.

If the list of words is: [“leopard”, “gorilla”, “dinosaur”, “cat”, “bee”] Then trying to complete “oar” would yield “leopard” and “dinosaur”, but not the others, because they match the regular expression ‘o.*a.*r’. Similar, in another application “djim” could expand to “django_migrations”.

The results are sorted by relevance, which is defined as the start position and the length of the match.

Notice that this is not really a tool to work around spelling mistakes, like what would be possible with difflib. The purpose is rather to have a quicker or more intuitive way to filter the given completions, especially when many completions have a common prefix.

Fuzzy algorithm is based on this post: https://blog.amjith.com/fuzzyfinder-in-10-lines-of-python

Parameters

- completer – A Completer instance.
- WORD – When True, use WORD characters.
- pattern –  Regex pattern which selects the characters before the cursor that are considered for the fuzzy matching.
- enable_fuzzy – (bool or Filter) Enabled the fuzzy behavior. For easily turning fuzzyness on or off according to a certain condition.

get_completions(
document: prompt_toolkit.document.Document,
complete_event:
prompt_toolkit.completion.base.CompleteEvent) ➞ 
Iterable[prompt_toolkit.completion.base.Completion]

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.

```python
class prompt_toolkit.completion.FuzzyWordCompleter(
    words: Union[List[str], Callable[[], List[str]]],
    meta_dict: Optional[Dict[str, str]] = None,
    WORD: bool = False
)
```

Fuzzy completion on a list of words.

(This is basically a WordCompleter wrapped in a FuzzyCompleter.)

Parameters

- **words** – List of words or callable that returns a list of words.
- **meta_dict** – Optional dict mapping words to their meta-information.
- **WORD** – When True, use WORD characters.

```python
def get_completions(
    document: prompt_toolkit.document.Document,
    complete_event: prompt_toolkit.completion.base.CompleteEvent
) -> Iterable[prompt_toolkit.completion.base.Completion]
```

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.

```python
class prompt_toolkit.completion.NestedCompleter(
    options: Dict[str, Optional[prompt_toolkit.completion.base.Completer]],
    ignore_case: bool = True
)
```

Completer which wraps around several other completers, and calls any the one that corresponds with the first word of the input.

By combining multiple NestedCompleter instances, we can achieve multiple hierarchical levels of autocompletion. This is useful when WordCompleter is not sufficient.

If you need multiple levels, check out the from_nested_dict classmethod.

```python
classmethod from_nested_dict(
    data: Mapping[str, Union[Any, Set[str], None, prompt_toolkit.completion.base.Completer]],
    ignore_case: bool = True
) -> prompt_toolkit.completion.nested.NestedCompleter
```

Create a NestedCompleter, starting from a nested dictionary data structure, like this:

```python
data = {
    'show': {
        'version': None,
        'interfaces': None,
        'clock': None,
        'ip': {'interface': {'brief'}}
    },
    'exit': None,
    'enable': None
}
```
The value should be *None* if there is no further completion at some point. If all values in the dictionary are None, it is also possible to use a set instead.

Values in this data structure can be a completers as well.

```python
```

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**

- `complete_event` -- `CompleteEvent` instance.

```python
class prompt_toolkit.completion.PathCompleter(only_directories: bool = False, get_paths: Optional[Callable[[], List[str]]] = None, file_filter: Optional[Callable[[str], bool]] = None, min_input_len: int = 0, expanduser: bool = 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.

```python
```

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**

- `complete_event` -- `CompleteEvent` instance.

```python
class prompt_toolkit.completion.ThreadedCompleter(completer: prompt_toolkit.completion.base.Completer)
```

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.
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.

Asynchronous generator of completions.

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-text. (This should map strings to strings or formatted text.)
- **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.
- **pattern** – Optional compiled regex for finding the word before the cursor to complete. When given, use this regex pattern instead of default one (see document._FIND_WORD_RE)
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
- `complete_event` – CompleteEvent instance.

```python
```

Return the common prefix for all completions.

```python
def merge_completers(completers: Sequence[prompt_toolkit.completion.base.Completer], deduplicate: bool = False) -> prompt_toolkit.completion.base.Completer
```

Combine several completers into one.

Parameters `deduplicate` – If True, wrap the result in a DeduplicateCompleter so that completions that would result in the same text will be deduplicated.

### 3.11.7 Document

The Document that implements all the text operations/querying.

```python
```

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

**property** char_before_cursor: str
Return character before the cursor or an empty string.

**property** current_char: str
Return character under cursor or an empty string.

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

**property** current_line_after_cursor: str
Text from the cursor until the end of the line.

**property** current_line_before_cursor: str
Text from the start of the line until the cursor.

**property** cursor_position: int
The document cursor position.
property cursor_position_col: int
    Current column. (0-based.)

property cursor_position_row: int
    Current row. (0-based.)

cut_selection() → Tuple[prompt_toolkit.document.Document,
    prompt_toolkit.clipboard.base.ClipboardData]
    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() → int
    Return number of empty lines at the end of the document.

end_of_paragraph(count: int = 1, after: bool = False) → int
    Return the end of the current paragraph. (Relative cursor position.)

find(sub: str, in_current_line: bool = False, include_current_position: bool = False, ignore_case: bool = False, count: int = 1) → Optional[int]
    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: str, ignore_case: bool = False) → List[int]
    Find all occurrences of the substring. Return a list of absolute positions in the document.

find_backwards(sub: str, in_current_line: bool = False, ignore_case: bool = False, count: int = 1) → Optional[int]
    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: bool = False, include_leading_whitespace: bool = False, include_trailing_whitespace: bool = False) → Tuple[int, int]
    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: str, right_ch: str, start_pos: Optional[int] = None) → Optional[int]
    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: str, right_ch: str, end_pos: Optional[int] = None) → Optional[int]
    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: Optional[int] = None, end_pos: Optional[int] = None) → int
    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: Callable[[str, bool], count: int = 1] → Optional[int]
    Look downwards for empty lines. Return the line index, relative to the current line.

find_next_word_beginning(WORD: bool = False) → Optional[int]
    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: bool = False, count: int = 1, WORD: bool = False*) → Optional[int]

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: Callable[[str], bool], count: int = 1*) → Optional[int]

Look upwards for empty lines. Return the line index, relative to the current line.

**find_previous_word_beginning**(*count: int = 1, WORD: bool = False*) → Optional[int]

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: int = 1, WORD: bool = False*) → Optional[int]

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: int = 1, WORD: bool = False, pattern: Optional[Pattern[str]] = None*) → Optional[int]

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: int*) → int

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: int = 1, preferred_column: Optional[int] = None*) → int

Return the relative cursor position (character index) where we would be if the user pressed the arrow-down button.

**get_cursor_left_position**(*count: int = 1*) → int

Relative position for cursor left.

**get_cursor_right_position**(*count: int = 1*) → int

Relative position for cursor right.

**get_cursor_up_position**(*count: int = 1, preferred_column: Optional[int] = None*) → int

Return the relative cursor position (character index) where we would be if the user pressed the arrow-up button.

**get_end_of_document_position**() → int

Relative position for the end of the document.

**get_end_of_line_position**() → int

Relative position for the end of this line.

**get_start_of_document_position**() → int

Relative position for the start of the document.

**get_start_of_line_position**(*after_whitespace: bool = False*) → int

Relative position for the start of this line.

**get_word_before_cursor**(*WORD: bool = False, pattern: Optional[Pattern[str]] = None*) → str

Give the word before the cursor. If we have whitespace before the cursor this returns an empty string.

**Parameters**

- **pattern** – (None or compiled regex). When given, use this regex pattern.
**get_word_under_cursor** *(WORD: bool = False) → str*
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: str) → bool*
True when this substring is found at the cursor position.

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

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

**property is_cursor_at_the_end:** bool
True when the cursor is at the end of the text.

**property is_cursor_at_the_end_of_line:** bool
True when the cursor is at the end of this line.

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

**property leading_whitespace_in_current_line:** str
The leading whitespace in the left margin of the current line.

**property line_count:** int
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.

**property lines:** List[str]
Array of all the lines.

**property lines_from_current:** List[str]
Array of the lines starting from the current line, until the last line.

**property on_first_line:** bool
True when we are at the first line.

**property on_last_line:** bool
True when we are at the last line.

Return a new Document instance which contains the result if we would paste this data at the current cursor position.

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

**property selection:** Optional[prompt_toolkit.selection.SelectionState]
SelectionState object.

**selection_range() → Tuple[int, int]**
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: int) → Optional[Tuple[int, int]]
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() → Iterable[Tuple[int, int]]
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: int = 1, before: bool = False) → int
Return the start of the current paragraph. (Relative cursor position.)

property text: str
The document text.

translate_index_to_position(index: int) → Tuple[int, int]
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: int, col: int) → int
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

prompt_toolkit.enums.DEFAULT_BUFFER = 'DEFAULT_BUFFER'
Name of the default buffer.

class prompt_toolkit.enums.EditingMode(value)
An enumeration.

prompt_toolkit.enums.SEARCH_BUFFER = 'SEARCH_BUFFER'
Name of the search buffer.

prompt_toolkit.enums.SYSTEM_BUFFER = 'SYSTEM_BUFFER'
Name of the system buffer.

3.11.9 History

Implementations for the history of a Buffer.

NOTE: 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.DummyHistory
History object that doesn’t remember anything.

append_string(string: str) → None
Add string to the history.

load_history_strings() → Iterable[str]
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.)

\[\text{store\_string} \ (\text{string: str} \rightarrow \text{None})\]
Store the string in persistent storage.

class prompt\_toolkit.history.FileHistory(filename: str)
History class that stores all strings in a file.

\[\text{load\_history\_strings} () \rightarrow \text{Iterable}[\text{str}]\]
This should be a generator that yields str instances.

\[\text{It should yield the most recent items first, because they are the most important} \ (\text{The history can already}
\text{be used, even when it’s only partially loaded.})\]

\[\text{store\_string} (\text{string: str} \rightarrow \text{None})\]
Store the string in persistent storage.

class prompt\_toolkit.history.History
Base History class.

This also includes abstract methods for loading/storing history.

\[\text{append\_string} (\text{string: str} \rightarrow \text{None})\]
Add string to the history.

\[\text{get\_strings} () \rightarrow \text{List}[\text{str}]\]
Get the strings from the history that are loaded so far. (In order. Oldest item first.)

\[\text{load} () \rightarrow \text{AsyncGenerator}[\text{str, None}]\]
Load the history and yield all the entries in reverse order (latest, most recent history entry first).

\[\text{This method can be called multiple times from the Buffer to repopulate the history when prompting for}
\text{a new input. So we are responsible here for both caching, and making sure that strings that were were}
\text{appended to the history will be incorporated next time this method is called.}\]

\[\text{abstract load\_history\_strings} () \rightarrow \text{Iterable}[\text{str}]\]
This should be a generator that yields str instances.

\[\text{It should yield the most recent items first, because they are the most important} \ (\text{The history can already}
\text{be used, even when it’s only partially loaded.})\]

\[\text{abstract store\_string} (\text{string: str} \rightarrow \text{None})\]
Store the string in persistent storage.

class prompt\_toolkit.history.InMemoryHistory(history\_strings: Optional[Sequence[str]] = None)
History class that keeps a list of all strings in memory.

In order to prepopulate the history, it’s possible to call either append\_string for all items or pass a list of strings
to \texttt{\_\_init\_\_} here.

\[\text{load\_history\_strings} () \rightarrow \text{Iterable}[\text{str}]\]
This should be a generator that yields str instances.

\[\text{It should yield the most recent items first, because they are the most important} \ (\text{The history can already}
\text{be used, even when it’s only partially loaded.})\]

\[\text{store\_string} (\text{string: str} \rightarrow \text{None})\]
Store the string in persistent storage.

class prompt\_toolkit.history.ThreadedHistory(history: prompt\_toolkit.history.History)
Wrapper around History implementations that run the load() 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.

```python
append_string(string: str) → None
Add string to the history.
```

```python
load() → AsyncGenerator[|str, None|
Like History.load(), but call `self.load_history_strings()` in a background thread.
```

```python
load_history_strings() → Iterable[|str|
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.)
```

```python
store_string(string: str) → None
Store the string in persistent storage.
```

### 3.11.10 Keys

```python
class prompt_toolkit.keys.Keys(value)
List of keys for use in key bindings.
```

Note that this is an “StrEnum”, all values can be compared against strings.

### 3.11.11 Style

Styling for prompt_toolkit applications.

```python
class prompt_toolkit.styles.AdjustBrightnessStyleTransformation(min_brightness: Union[Callable[|[]|, float], float] = 0.0, max_brightness: Union[Callable[|[]|, float], float] = 1.0)
```

Adjust the brightness to improve the rendering on either dark or light backgrounds.

For dark backgrounds, it’s best to increase `min_brightness`. For light backgrounds it’s best to decrease `max_brightness`. Usually, only one setting is adjusted.

This will only change the brightness for text that has a foreground color defined, but no background color. It works best for 256 or true color output.

**Note:** Notice that there is no universal way to detect whether the application is running in a light or dark terminal. As a developer of an command line application, you’ll have to make this configurable for the user.

**Parameters**

- `min_brightness` – Float between 0.0 and 1.0 or a callable that returns a float.
- `max_brightness` – Float between 0.0 and 1.0 or a callable that returns a float.

```python
invalidation_hash() → Hashable
When this changes, the cache should be invalidated.
```
**transform_attrs** *(attrs: prompt_toolkit.styles.base.Attrs) → prompt_toolkit.styles.base.Attrs*

Take an *Attrs* object and return a new *Attrs* object.

Remember that the color formats can be either “ansi...” or a 6 digit lowercase hexadecimal color (without ‘#’ prefix).

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

**property bgcolor**

Alias for field number 1

**property blink**

Alias for field number 6

**property bold**

Alias for field number 2

**property color**

Alias for field number 0

**property hidden**

Alias for field number 8

**property italic**

Alias for field number 5

**property reverse**

Alias for field number 7

**property strike**

Alias for field number 4

**property underline**

Alias for field number 3

**class** *prompt_toolkit.styles.BaseStyle*

Abstract base class for prompt_toolkit styles.


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.

**abstract invalidation_hash**() → Hashable

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.

**abstract property style_rules:** List[Tuple[str, str]]

The list of style rules, used to create this style. (Required for *DynamicStyle* and *_MergedStyle* to work.)
class prompt_toolkit.styles.ConditionalStyleTransformation(style_transformation: 
prompt_toolkit.styles.style_transformation.StyleTransformation, 
filter: 
Union[prompt_toolkit.filters.base.Filter, bool])

Apply the style transformation depending on a condition.

invalidation_hash() → Hashable
When this changes, the cache should be invalidated.

transform_attrs(atts: prompt_toolkit.styles.base.Attrs) → prompt_toolkit.styles.base.Attrs
Take an Attrs object and return a new Attrs object.

Remember that the color formats can be either “ansi...” or a 6 digit lowercase hexadecimal color (without ‘#’ prefix).

class prompt_toolkit.styles.DummyStyle
A style that doesn’t style anything.

gattrs_for_style_str(style_str: str, default: prompt_toolkit.styles.base.Attrs = Attrs(color=", 
bbgcolor=", bold=False, underline=False, strike=False, italic=False, 
blink=False, reverse=False, hidden=False)) → prompt_toolkit.styles.base.Attrs

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() → Hashable
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.

property style_rules: List[Tuple[str, str]]
The list of style rules, used to create this style. (Required for DynamicStyle and _MergedStyle to work.)

class prompt_toolkit.styles.DummyStyleTransformation
Don’t transform anything at all.

invalidation_hash() → Hashable
When this changes, the cache should be invalidated.

transform_attrs(atts: prompt_toolkit.styles.base.Attrs) → prompt_toolkit.styles.base.Attrs
Take an Attrs object and return a new Attrs object.

Remember that the color formats can be either “ansi...” or a 6 digit lowercase hexadecimal color (without ‘#’ prefix).

class prompt_toolkit.styles.DynamicStyle(get_style: Callable[[], 
Optional[prompt_toolkit.styles.base.BaseStyle]])

Style class that can dynamically returns an other Style.

Parameters get_style – Callable that returns a Style instance.

gattrs_for_style_str(style_str: str, default: prompt_toolkit.styles.base.Attrs = Attrs(color=", 
bbgcolor=", bold=False, underline=False, strike=False, italic=False, 
blink=False, reverse=False, hidden=False)) → prompt_toolkit.styles.base.Attrs

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() → Hashable

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.

**property style_rules: List[Tuple[str, str]]**

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

**class** prompt_toolkit.styles.DynamicStyleTransformation(get_style_transformation: Callable[[], Optional[prompt_toolkit.styles.style_transformation.StyleTransformation]])

StyleTransformation class that can dynamically returns any StyleTransformation.

**Parameters**

get_style_transformation – Callable that returns a StyleTransformation instance.

invalidation_hash() → Hashable

When this changes, the cache should be invalidated.

transform_attrs(attrs: prompt_toolkit.styles.base.Attrs) → prompt_toolkit.styles.base.Attrs

Take an Attrs object and return a new Attrs object.

Remember that the color formats can be either “ansi...” or a 6 digit lowercase hexadecimal color (without ‘#’ prefix).

**class** prompt_toolkit.styles.Priority(value)

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.)

**class** prompt_toolkit.styles.Style(style_rules: List[Tuple[str, str]])

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:

```python
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: Dict[str, str], priority: prompt_toolkit.styles.style.Priority = Priority.DICT_KEY_ORDER) → prompt_toolkit.styles.style.Style

Parameters

• style_dict – Style dictionary.

• priority – Priority value.

get_attrs_for_style_str(style_str: str, default: prompt_toolkit.styles.base.Attrs = Attrs(color="", bgcolor="", bold=False, underline=False, strike=False, italic=False, blink=False, reverse=False, hidden=False)) → prompt_toolkit.styles.base.Attrs

GetAttrs for the given style string.

invalidation_hash() → Hashable

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.

property style_rules: List[Tuple[str, str]]

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

class prompt_toolkit.styles.StyleTransformation

Base class for any style transformation.

invalidation_hash() → Hashable

When this changes, the cache should be invalidated.

abstract transformAttrs(attrs: prompt_toolkit.styles.base.Attrs) → prompt_toolkit.styles.base.Attrs

Take an Attrs object and return a new Attrs object.

Remember that the color formats can be either “ansi…” or a 6 digit lowercase hexadecimal color (without ‘#’ prefix).

class prompt_toolkit.styles.SwapLightAndDarkStyleTransformation

Turn dark colors into light colors and the other way around.

This is meant to make color schemes that work on a dark background usable on a light background (and the other way around).

Notice that this doesn’t swap foreground and background like “reverse” does. It turns light green into dark green and the other way around. Foreground and background colors are considered individually.

Also notice that when <reverse> is used somewhere and no colors are given in particular (like what is the default for the bottom toolbar), then this doesn’t change anything. This is what makes sense, because when the ‘default’ color is chosen, it’s what works best for the terminal, and reverse works good with that.

transformAttrs (attrs: prompt_toolkit.styles.base.Attrs) → prompt_toolkit.styles.base.Attrs

Return the Attrs used when opposite luminosity should be used.

prompt_toolkit.styles.merge_style_transformations(style_transformations: Sequence[prompt_toolkit.styles.style_transformation.StyleTransformation]) → prompt_toolkit.styles.style_transformation.StyleTransformation

Merge multiple transformations together.

prompt_toolkit.styles.merge_styles(styles: List[prompt_toolkit.styles.base.BaseStyle]) → prompt_toolkit.styles.style._MergedStyle

Merge multiple Style objects.

prompt_toolkit.styles.pygments_token_to_classname(token: Token) → str

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.)

\[
\text{prompt_toolkit.styles.style_from_pygments_cls}(\text{pygments_style_cls: Type[PygmentsStyle]}) \rightarrow \text{prompt_toolkit.styles.style.Style}
\]

Shortcut to create a \textit{Style} instance from a Pygments style class and a style dictionary.

Example:

```python
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**

- \texttt{pygments_style_cls} – Pygments style class to start from.

\[
\text{prompt_toolkit.styles.style_from_pygments_dict}(\text{pygments_dict: Dict[Token, str]} \rightarrow \text{prompt_toolkit.styles.style.Style}
\]

Create a \textit{Style} instance from a Pygments style dictionary. (One that maps Token objects to style strings.)

### 3.11.12 Shortcuts

**class** `prompt_toolkit.shortcuts.CompleteStyle(value)`

How to display autocompletions for the prompt.


Progress bar context manager.

**Usage**

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

**Parameters**

- \texttt{title} – Text to be displayed above the progress bars. This can be a callable or formatted text as well.
- \texttt{formatters} – List of \textit{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.
- **keyBindings** – `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.
class prompt_toolkit.shortcuts.PromptSession


Chapter 3: Table of contents
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:

```python
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.
- **style_transformation** – `StyleTransformation` instance.
- **swap_light_and_dark_colors** – `bool` or `Filter`. When enabled, apply `SwapLightAndDarkStyleTransformation`. This is useful for switching between dark and light terminal backgrounds.
- **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 `prompt_width`, `line_number` and `wrap_count` as input and returns formatted text. When this is `None` (the default), then `prompt_width` spaces will be used.

• `complete_style` – `CompleteStyle.COLUMN`, `CompleteStyle.MULTI_COLUMN` or `CompleteStyle.READLINE_LIKE`.

• `mouse_support` – `bool` or `Filter` to enable mouse support.

• `placeholder` – Text to be displayed when no input has been given yet. Unlike the `default` parameter, this won’t be returned as part of the output ever. This can be formatted text or a callable that returns formatted text.

• `refresh_interval` – (number; in seconds) When given, refresh the UI every so many seconds.

• `input` – `Input` object. (Note that the preferred way to change the input/output is by creating an `AppSession`.)

• `output` – `Output` object.
prompt

```
message: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]], Callable[[[], Any]]] = None, *

edit mode: Optional[prompt_toolkit.enums.EditingMode] = None, refresh_interval: Optional[Callable[[prompt_toolkit.mouse_events.MouseEvent], None]] = None, key_bindings: Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase] = None, first_react: Optional[Callable[[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase], None]] = None, key_bindings: Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key Binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on_cancel: Optional[Callable[[prompt_toolkit.key_binding.keybindings.KeyBindingsBase], None]] = None, escape_on
```

The first set of arguments is a subset of the `PromptSession` class itself. For these, passing in `None` will keep the current values that are active in the session. Passing in a value will set the attribute for the session, which means that it applies to the current, but also to the next prompts.

Note that in order to erase a Completer, Validator or AutoSuggest, you can’t use `None`. Instead pass in a `DummyCompleter`, `DummyValidator` or `DummyAutoSuggest` instance respectively. For a Lexer you can pass in an empty `SimpleLexer`.

3.11. Reference 129
Additional arguments, specific for this prompt:

**Parameters**

- **default** – The default input text to be shown. (This can be edited by the user).
- **accept_default** – When True, automatically accept the default value without allowing the user to edit the input.
- **pre_run** – Callable, called at the start of `Application.run`.
- **in_thread** – Run the prompt in a background thread; block the current thread. This avoids interference with an event loop in the current thread. Like `Application.run(in_thread=True)`.

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

```
prompt_toolkit.shortcuts.button_dialog(title: Optional[Union[str, MagicFormattedText, 
    List[Union[Tuple[str, str], Tuple[str, str, 
    Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], 
    Callable[[], Any]]] = ", 
    text: Optional[Union[str, 
    MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str, 
    Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], 
    Callable[[], Any]]] = "", 
    buttons: List[Tuple[str, 
    prompt_toolkit.shortcuts.dialogs._T]] = [], 
    style: Optional[prompt_toolkit.shortcuts.prompt.PromptSession[None]]
    ) -> 
    prompt_toolkit.application.application.Application

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

prompt_toolkit.shortcuts.clear() -> None
Clear the screen.

prompt_toolkit.shortcuts.clear_title() -> None
Erase the current title.

prompt_toolkit.shortcuts.confirm(message: str = 'Confirm?', suffix: str = ' (y/n) ') -> bool
Display a confirmation prompt that returns True/False.

prompt_toolkit.shortcuts.create_confirm_session(message: str, suffix: str = ' (y/n) ') ->
    prompt_toolkit.shortcuts.prompt.PromptSession[bool]
Create a PromptSession object for the 'confirm' function.

prompt_toolkit.shortcuts.input_dialog(title: Optional[Union[str, MagicFormattedText, 
    List[Union[Tuple[str, str], Tuple[str, str, 
    Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], 
    Callable[[], Any]]] = ", 
    text: Optional[Union[str, 
    MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str, 
    Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], 
    Callable[[], Any]]] = "", 
    ok_text: str = 'OK', cancel_text: str = 
    'Cancel', completer: Optional[prompt_toolkit.completion.base.Completer] = None, 
    validator: Optional[prompt_toolkit.validation.Validator] = Noe, 
    password: Union[prompt_toolkit.filters.base.Filter, bool] = False, 
    style: Optional[prompt_toolkit.styles.base.BaseStyle] = None) -> 
    prompt_toolkit.application.application.Application

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

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


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:

```python
print_formatted_text(HTML('<i>Some italic text</i> <ansired>This is red!</ansired>'), style=Style.from_dict({'hello': '#ff0066', 'world': '#884444 italic'}))
```

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

```python
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.

If a prompt_toolkit Application is currently running, this will always print above the application or prompt (sim-
similar to `patch_stdout`). So, `print_formatted_text` will erase the current application, print the text, and render the application again.

**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).

**Parameters**

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

Display the prompt.
The first set of arguments is a subset of the `PromptSession` class itself. For these, passing in `None` will keep the current values that are active in the session. Passing in a value will set the attribute for the session, which means that it applies to the current, but also to the next prompts.

Note that in order to erase a Completer, Validator or AutoSuggest, you can’t use `None`. Instead pass in a `DummyCompleter`, `DummyValidator` or `DummyAutoSuggest` instance respectively. For a Lexer you can pass in an empty `SimpleLexer`.

Additional arguments, specific for this prompt:

**Parameters**

- **default** – The default input text to be shown. (This can be edited by the user).
- **accept_default** – When `True`, automatically accept the default value without allowing the user to edit the input.
- **pre_run** – Callable, called at the start of `Application.run`
- **in_thread** – Run the prompt in a background thread; block the current thread. This avoids interference with an event loop in the current thread. Like `Application.run(in_thread=True)`.

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

```
prompt_toolkit.shortcuts.radiolist_dialog(title: Optional[Union[str, MagicFormattedText,
            List[Union[Tuple[str, str], Tuple[str, str,
            Callable[[prompt_toolkit.mouse_events.MouseEvent],
            None]]]], Callable[[str, None]]] = '',
            text: Optional[Union[str,
            MagicFormattedText, List[Union[Tuple[str, str],
            Tuple[str, str,
            Callable[[prompt_toolkit.mouse_events.MouseEvent],
            None]]]], Callable[[str, None]]] = '',
            ok_text: str = 'Ok',
            cancel_text: str = 'Cancel',
            values: Optional[Sequence[Tuple[prompt_toolkit.shortcuts.dialogs._T,
            Optional[Union[str, MagicFormattedText,
            List[Union[Tuple[str, str], Tuple[str, str,
            Callable[[prompt_toolkit.mouse_events.MouseEvent],
            None]]], Callable[[str, None]]]]
            ]] = None,
            default: Optional[prompt_toolkit.shortcuts.dialogs._T] = None,
            style: Optional[prompt_toolkit.styles.base.BaseStyle] = None) →
prompt_toolkit.application.application.Application[prompt_toolkit.shortcuts.dialogs._T]
```

Display a simple list of element the user can choose amongst.

Only one element can be selected at a time using Arrow keys and Enter. The focus can be moved between the list and the Ok/Cancel button with tab.

```
prompt_toolkit.shortcuts.set_title(text: str) → None
```

Set the terminal title.

```
prompt_toolkit.shortcuts.yes_no_dialog(title: Optional[Union[str, MagicFormattedText,
            List[Union[Tuple[str, str], Tuple[str, str,
            Callable[[prompt_toolkit.mouse_events.MouseEvent],
            None]]]], Callable[[str, None]]] = '',
            text: Optional[Union[str,
            MagicFormattedText, List[Union[Tuple[str, str],
            Tuple[str, str,
            Callable[[prompt_toolkit.mouse_events.MouseEvent],
            None]]], Callable[[str, None]]]] = '',
            yes_text: str = 'Yes',
            no_text: str = 'No',
            style: Optional[prompt_toolkit.styles.base.BaseStyle] = None) →
prompt_toolkit.application.application.Application[bool]
```

Display a Yes/No dialog. Return a boolean.
Formatter classes for the progress bar. Each progress bar consists of a list of these formatters.

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Bar:
    start: str = '\',
    end: str = '\',
    sym_a: str = '\',
    sym_b: str = '>',
    sym_c: str = '\',
    unknown: str = '#')
```

Display the progress bar itself.

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Formatter:
    Base class for any formatter.
```

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.IterationsPerSecond:
    Display the iterations per second.
```

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Label:
    width: Union[None, int, prompt_toolkit.layout.dimension.Dimension, Callable[[], Any]] = None,
    suffix: str = '\n')
```

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.

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Percentage:
    Display the progress as a percentage.
```

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Progress:
    Display the progress as text. E.g. “8/20”
```

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Rainbow:
    formatter:
        prompt_toolkit.shortcuts.progress_bar.formatters.Formatter
```

For the fun. Add rainbow colors to any of the other formatters.

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.SpinningWheel:
    Display a spinning wheel.
```

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.Text:
    text: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[], None]]], Callable[[], Any]]],
    style: str = '\n')
```

Display plain text.

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.TimeElapsed:
    Display the elapsed time.
```

```python
class prompt_toolkit.shortcuts.progress_bar.formatters.TimeLeft:
    Display the time left.
```

`prompt_toolkit.shortcuts.progress_bar.formatters.create_default_formatters() → List[prompt_toolkit.shortcuts.progress_bar.formatters.Formatter]`

Return the list of default formatters.
3.11.13 Validation

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

```python
```

Validator that can be switched on/off according to a filter. (This wraps around another validator.)

```python
    Validate the input. If invalid, this should raise a ValidationError.
```

Parameters

```python
class prompt_toolkit.validation.DummyValidator
```

Validator class that accepts any input.

```python
    Validate the input. If invalid, this should raise a ValidationError.
```

Parameters

class prompt_toolkit.validation.DynamicValidator(get_validator: Callable[[], Optional[prompt_toolkit.validation.Validator]])
```

Validator class that can dynamically returns any Validator.

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

```python
    Validate the input. If invalid, this should raise a ValidationError.
```

Parameters

```python
    Run the validate function in a thread.
```

exception prompt_toolkit.validation.ValidationError(cursor_position: int = 0, message: str = '')
```

Error raised by `Validator.validate()`.

Parameters
  • cursor_position – The cursor position where the error occurred.
  • message – Text.

```python
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`.

```python
classmethod from_callable(validate_func: Callable[[str], bool], error_message: str = 'Invalid input', move_cursor_to_end: bool = False) -> prompt_toolkit.validation.Validator
```

Create a validator from a simple validate callable. E.g.:

```python
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.

```python
abstract validate(document: prompt_toolkit.document.Document) -> None
```

Validate the input. If invalid, this should raise a `ValidationErrror`.

**Parameters**


```python
async validate_async(document: prompt_toolkit.document.Document) -> None
```

Return a `Future` which is set when the validation is ready. This function can be overloaded in order to provide an asynchronous implementation.

### 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`.

```python
class prompt_toolkit.auto_suggest.AutoSuggest
```

Base class for auto suggestion implementations.

```python
abstract get_suggestion(buffer: Buffer, document: prompt_toolkit.document.Document) ->
    Optional[prompt_toolkit.auto_suggest.Suggestion]
```

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.
Optional[prompt_toolkit.auto_suggest.Suggestion]
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.AutoSuggestFromHistory
Give suggestions based on the lines in the history.

Optional[prompt_toolkit.auto_suggest.Suggestion]
Return None or a Suggestion instance.

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

Parameters
• buffer – The Buffer instance.
• document – The Document instance.

class prompt_toolkit.auto_suggest.ConditionalAutoSuggest(auto_suggest:
prompt_toolkit.auto_suggest.AutoSuggest,
filter: Union[bool,
prompt_toolkit.filters.base.Filter])
Auto suggest that can be turned on and off according to a certain condition.

Optional[prompt_toolkit.auto_suggest.Suggestion]
Return None or a Suggestion instance.

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

Parameters
• buffer – The Buffer instance.
• document – The Document instance.

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

Optional[prompt_toolkit.auto_suggest.Suggestion]
Return None or a Suggestion instance.

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

Parameters
• buffer – The Buffer instance.
• document – The Document instance.
class prompt_toolkit.auto_suggest.DynamicAutoSuggest(get_auto_suggest: Callable[[], Optional[prompt_toolkit.auto_suggest.AutoSuggest]])

Validator class that can dynamically returns any Validator.

Parameters

- get_validator – Callable that returns a Validator instance.


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.


Run the get_suggestion function in a thread.

class prompt_toolkit.auto_suggest.Suggestion(text: str)

Suggestion returned by an auto-suggest algorithm.

Parameters

- text – The suggestion text.

class prompt_toolkit.auto_suggest.ThreadedAutoSuggest(auto_suggest: prompt_toolkit.auto_suggest.AutoSuggest)

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.)


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.


Run the get_suggestion function in a thread.
3.11.15 Renderer

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

```python
```

Typical usage:

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

clear() → None
Clear screen and go to 0,0

erase(leave_alternate_screen: bool = True) → None
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.

property height_is_known: bool
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.)

property last_rendered_screen: Optional[prompt_toolkit.layout.screen.Screen]
The `Screen` class that was generated during the last rendering. This can be `None`.

render(app: Application[Any], layout: Layout, is_done: bool = False) → None
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: int) → None
To be called when we know the absolute cursor position. (As an answer of a “Cursor Position Request” response.)

request_absolute_cursor_position() → None
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.)

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

async wait_for_cpr_responses(timeout: int = 1) → None
Wait for a CPR response.

property waiting_for_cpr: bool
Waiting for CPR flag. True when we send the request, but didn’t got a response.
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.DynamicLexer**

`DynamicLexer` is a Lexer class that can dynamically returns any Lexer.

**Parameters**

- `get_lexer` – Callable that returns a `Lexer` instance.

**invalidation_hash**

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

**lex_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.

**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.Lexer**

Base class for all lexers.

**invalidation_hash**

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

**abstract lex_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.

**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.PygmentsLexer**

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.

Create a Lexer from a filename.

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: str)
Synchronize by starting at a line that matches the given regex pattern.

classmethod `from_pygments_lexer_cls`(lexer_cls: PygmentsLexerCls) → RegexSync
Create a RegexSync instance for this Pygments lexer class.

Scan backwards, and find a possible position to start.

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

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

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.SyncFromStart
Always start the syntax highlighting from the beginning.

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

Parameters

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

**class** prompt_toolkit.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.

**abstract get_sync_start_position**

```python
```

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

### 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

The layout class itself

class prompt_toolkit.layout.Layout(
    container: Union[prompt_toolkit.layout.containers.Container,
                     MagicContainer],
    focused_element: Optional[Union[str,
                                   prompt_toolkit.buffer.Buffer,
                                   prompt_toolkit.layout.controls.UIManager,
                                   prompt_toolkit.layout.containers.Container, MagicContainer]] = 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.)

property buffer_has_focus: bool
    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.)

property current_buffer: Optional[prompt_toolkit.buffer.Buffer]
    The currently focused Buffer or None.

property current_control: prompt_toolkit.layout.controls.UIManager
    Get the UIManager to currently has the focus.

property current_window: prompt_toolkit.layout.containers.Window
    Return the Window object that is currently focused.

find_all_windows() → Generator[prompt_toolkit.layout.containers.Window, None, None]
    Find all the UIManager objects in this layout.

focus(value: Union[str, prompt_toolkit.buffer.Buffer, prompt_toolkit.layout.controls.UIManager,
                   prompt_toolkit.layout.containers.Container, MagicContainer]) → None
    Focus the given UI element.

value can be either:

• a UIManager
• 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() → None
    Give the focus to the last focused control.

focus_next() → None
    Focus the next visible/focusable Window.

focus_previous() → None
    Focus the previous visible/focusable Window.

get_buffer_by_name(buffer_name: str) → Optional[prompt_toolkit.buffer.Buffer]
    Look in the layout for a buffer with the given name. Return None when nothing was found.
get_focused_windows() \rightarrow \text{Iterable}[\text{Window}]

Return all the Window objects which are focusable (in the ‘modal’ area).

get_parent(container: Container) \rightarrow \text{Optional[Container]}

Return the parent container for the given container, or None, if it wasn’t found.

get_visible_focused_windows() \rightarrow \text{List[Window]}

Return a list of Window objects that are focusable.

has_focus(value: UIControl or Window instance) \rightarrow \text{bool}

Check whether the given control has the focus. :param value: UIControl or Window instance.

property is_searching: bool

True if we are searching right now.

property previous_control: UIControl

Get the UIControl to previously had the focus.

property search_target_buffer_control:

Optional[BufferControl]

Return the BufferControl in which we are searching or None.

update_parents_relations() \rightarrow \text{None}

Update child->parent relationships mapping.

walk() \rightarrow \text{Iterable[Container]}

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

walk_through_modal_area() \rightarrow \text{Iterable[Container]}

Walk through all the containers which are in the current ‘modal’ part of the layout.

class InvalidLayoutError

class walk(container: Container, skip_hidden: bool = False)

Walk through layout, starting at this container.

Containers

class Container

Base class for user interface layout.

abstract get_children() \rightarrow \text{List[Container]}

Return the list of child Container objects.

get_key_bindings() \rightarrow \text{Optional[KeyBindingsBase]}

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() \rightarrow \text{bool}

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

abstract preferred_height(width: int, max_available_height: int) \rightarrow \text{Dimension}

Return a Dimension that represents the desired height for this container.

abstract preferred_width(max_available_width: int) \rightarrow \text{Dimension}

Return a Dimension that represents the desired width for this container.
abstract reset() → None
   Reset the state of this container and all the children. (E.g. reset scroll offsets, etc…)


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.


Several layouts, one stacked above/under the other.

```
+-------------------------+
|                        |
+-------------------------+
|                        |
+-------------------------+
```

By default, this doesn’t display a horizontal line between the children, but if this is something you need, then create a HSplit as follows:

```
HSplit(children=[ ... ], padding_char='-',
        padding=1, padding_style='#ffff00')
```

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.
   • align – VerticalAlign value.
   • width – When given, use this width instead of looking at the children.
• **height** – When given, use this height 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.

• **padding** – (Dimension or int), size to be used for the padding.

• **padding_char** – Character to be used for filling in the padding.

• **padding_style** – Style to be applied to the padding.

```python
preferred_height(width: int, max_available_height: int) \rightarrow prompt_toolkit.layout.dimension.Dimension
```

Return a **Dimension** that represents the desired height for this container.

```python
preferred_width(max_available_width: int) \rightarrow prompt_toolkit.layout.dimension.Dimension
```

Return a **Dimension** that represents the desired width for this container.

```python
reset() \rightarrow None
```

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

```python
```

Render the prompt to a Screen instance.

**Parameters**

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

```python
```

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

```
+--------+--------+
|   |   |   |
|   |   |   |
+--------+--------+
```

By default, this doesn’t display a vertical line between the children, but if this is something you need, then create a HSplit as follows:

```python
VSplit(children=[ ... ], padding_char="|",
padding=1, padding_style="#ffffff")
```
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.
- **align** – *HorizontalAlign* value.
- **width** – When given, use this width instead of looking at the children.
- **height** – When given, use this height 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.
- **padding** – *(Dimension or int)*, size to be used for the padding.
- **padding_char** – Character to be used for filling in the padding.
- **padding_style** – Style to applied to the padding.

```python
preferred_height(width: int, max_available_height: int) → prompt_toolkit.layout.dimension.Dimension
```

Return a *Dimension* that represents the desired height for this container.

```python
preferred_width(max_available_width: int) → prompt_toolkit.layout.dimension.Dimension
```

Return a *Dimension* that represents the desired width for this container.

```python
reset() → None
```

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

```python
```

Render the prompt to a *Screen* instance.

**Parameters**

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

```python
```

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

Example Usage:

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

**get_children() → List[prompt_toolkit.layout.containers.Container]**
Return the list of child *Container* objects.

**get_key_bindings() → Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase]**
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() → bool**
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: int, max_available_height: int) → prompt_toolkit.layout.dimension.Dimension**
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.)

**preferred_width(max_available_width: int) → prompt_toolkit.layout.dimension.Dimension**
Return a *Dimension* that represents the desired width for this container.

**reset() → None**
Reset the state of this container and all the children. (E.g. reset scroll offsets, etc…)

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.


*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.

```python
```

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. (This can be a callable that returns a string.)

• **char** – (string) Character to be used for filling the background. This can also be a callable that returns a character.

• **get_line_prefix** – None or a callable that returns formatted text to be inserted before a line. It takes a line number (int) and a wrap_count and returns formatted text. This can be used for implementation of line continuations, things like Vim “breakindent” and so on.
get_children() → List[prompt_toolkit.layout.containers.Container]
Return the list of child Container objects.

get_key_bindings() → Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase]
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.

preferred_height(width: int, max_available_height: int) → prompt_toolkit.layout.dimension.Dimension
Calculate the preferred height for this window.

preferred_width(max_available_width: int) → prompt_toolkit.layout.dimension.Dimension
Calculate the preferred width for this window.

reset() → None
Reset the state of this container and all the children. (E.g. reset scroll offsets, etc…)

write_to_screen(screen: prompt_toolkit.layout.screen.Screen, mouse_handlers:
prompt_toolkit.layout.mouse_handlers.MouseHandlers, write_position:
prompt_toolkit.layout.screen.WritePosition, parent_style: str, erase_bg: bool, z_index:
Optional[int]) → None
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(value)
Alignment of the Window content.

Note that this is different from HorizontalAlign and VerticalAlign, which are used for the alignment of the child
containers in respectively VSplit and HSplit.

class prompt_toolkit.layout.ConditionalContainer(content:
Union[prompt_toolkit.layout.containers.Container,
MagicContainer], filter:
Union[prompt_toolkit.filters.base.Filter, bool])
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.

get_children() → List[prompt_toolkit.layout.containers.Container]
Return the list of child Container objects.

preferred_height(width: int, max_available_height: int) → prompt_toolkit.layout.dimension.Dimension
Return a Dimension that represents the desired height for this container.

preferred_width(max_available_width: int) → prompt_toolkit.layout.dimension.Dimension
Return a Dimension that represents the desired width for this container.

reset() → None
Reset the state of this container and all the children. (E.g. reset scroll offsets, etc…)

write_to_screen(screen: prompt_toolkit.layout.screen.Screen, mouse_handlers:
prompt_toolkit.layout.mouse_handlers.MouseHandlers, write_position:
prompt_toolkit.layout.screen.WritePosition, parent_style: str, erase_bg: bool, z_index:
Optional[int]) → None
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.DynamicContainer(get_container: Callable[],
   Union[prompt_toolkit.layout.containers.Container,
        MagicContainer]])

Container class that dynamically returns any Container.

Parameters get_container – Callable that returns a Container instance or any widget with a `__pt_container__` method.

get_children() → List[prompt_toolkit.layout.containers.Container]
Return the list of child Container objects.

get_key_bindings() → Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase]
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() → bool
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: int, max_available_height: int) → prompt_toolkit.layout.dimension.Dimension
Return a Dimension that represents the desired height for this container.

preferred_width(max_available_width: int) → prompt_toolkit.layout.dimension.Dimension
Return a Dimension that represents the desired width for this container.

reset() → None
Reset the state of this container and all the children. (E.g. reset scroll offsets, etc…)

write_to_screen(screen: prompt_toolkit.layout.screen.Screen, mouse_handlers:
   prompt_toolkit.layout.mouse_handlers.MouseHandlers, write_position:
   prompt_toolkit.layout.screen.WritePosition, parent_style: str, erase_bg: bool, z_index:
   Optional[int]) → None
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.

Container widget that exposes a larger virtual screen to its content and displays it in a vertical scrollbale region. Typically this is wrapped in a large HSplit container. Make sure in that case to not specify a height dimension of the HSplit, so that it will scale according to the content.

Note: If you want to display a completion menu for widgets in this ScrollablePane, then it’s still a good practice to use a FloatContainer with a CompletionsMenu in a Float at the top-level of the layout hierarchy, rather then nesting a FloatContainer in this ScrollablePane. (Otherwise, it’s possible that the completion menu is clipped.)

Parameters

- **content** – The content container.
- **scroll_offset** – Try to keep the cursor within this distance from the top/bottom (left/right offset is not used).
- **keep_cursor_visible** – When True, automatically scroll the pane so that the cursor (of the focused window) is always visible.
- **keep_focused_window_visible** – When True, automatically scroll the pane so that the focused window is visible, or as much visible as possible if it doesn’t completely fit the screen.
- **max_available_height** – Always constraint the height to this amount for performance reasons.
- **width** – When given, use this width instead of looking at the children.
- **height** – When given, use this height instead of looking at the children.
- **show_scrollbar** – When True display a scrollbar on the right.

get_children() → List[prompt_toolkit.layout.containers.Container]

Return the list of child Container objects.

get_key_bindings() → Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase]

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() → bool

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: int, max_available_height: int) → prompt_toolkit.layout.dimension.Dimension

Return a Dimension that represents the desired height for this container.

preferred_width(max_available_width: int) → prompt_toolkit.layout.dimension.Dimension

Return a Dimension that represents the desired width for this container.

reset() → None

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


Render scrollable pane content.

This works by rendering on an off-screen canvas, and copying over the visible region.

class prompt_toolkit.layout.ScrollOffsets(top: Union[int, Callable[[], int]] = 0, bottom: Union[int, Callable[[], int]] = 0, left: Union[int, Callable[[], int]] = 0, right: Union[int, Callable[[], int]] = 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: int, style: str = 'class:color-column')

Column for a Window to be colored.

class prompt_toolkit.layout.to_container(container: Union[prompt_toolkit.layout.containers.Container, MagicContainer])

Make sure that the given object is a Container.

class prompt_toolkit.layout.to_window(container: Union[prompt_toolkit.layout.containers.Container, MagicContainer])

Make sure that the given argument is a Window.

class prompt_toolkit.layout.is_container(value: object)

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

class prompt_toolkit.layout.HorizontalAlign(value)

Alignment for VSplit.

class prompt_toolkit.layout.VerticalAlign(value)

Alignment for HSplit.
Controls

class prompt_toolkit.layout.BufferControl(buffer: Optional[prompt_toolkit.buffer.Buffer] = None,
input_processors:
Optional[List[prompt_toolkit.layout.processors.Processor]] = None,
include_default_input_processors: bool = True,
lexer:
Optional[prompt_toolkit.lexers.base.Lexer] = None,
preview_search: Union[prompt_toolkit.filters.base.Filter,
bool] = False,
focusable:
Union[prompt_toolkit.filters.base.Filter, bool] = True,
search_buffer_control: Union[None, SearchBufferControl,
Callable[[], SearchBufferControl]] = None,
menu_position: Optional[Callable[[], Optional[int]]] = None,
focus_on_click: Union[None, SearchBufferControl,
Callable[[], SearchBufferControl]] = False,
key_bindings: Optional[KeyBindingsBase] = 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: int, height: int, preview_search: bool = False) →
prompt_toolkit.layout.controls.UIContent

Create a UIContent.

get_invalidate_events() → Iterable[Event[object]]

Return the Window invalidate events.

get_key_bindings() → Optional[KeyBindingsBase]

When additional key bindings are given. Return these.

is_focussable() → bool

Tell whether this user control is focussable.

mouse_handler(mouse_event: prompt_toolkit.mouse_events.MouseEvent) → NotImplementedOrNone

Mouse handler for this control.

move_cursor_down() → None

Request to move the cursor down. This happens when scrolling down and the cursor is completely at the top.

move_cursor_up() → None

Request to move the cursor up.
preferred_width(max_available_width: int) → Optional[int]

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.

property search_state: prompt_toolkit.search.SearchState

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:

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.)

create_content(width: int, height: int) → prompt_toolkit.layout.controls.UIContent

Generate the content for this user control.

Returns a UIContent instance.

is_focusable() → bool

Tell whether this user control is focusable.

class prompt_toolkit.layout.FormattedTextControl:

Control that displays formatted text. This can be either plain text, an HTML object an ANSI object, a list of (style_str, text) tuples or a callable that takes no argument and returns one of those, 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:

3.11. Reference
• 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.

```python
class UIControl
    Base class for all user interface controls.

    abstract create_content(width: int, height: int) → prompt_toolkit.layout.controls.UIContent
        Generate the content for this user control.
        Returns a `UIContent` instance.
```

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.

```python
class UIControl
    Base class for all user interface controls.

    abstract create_content(width: int, height: int) → prompt_toolkit.layout.controls.UIContent
        Generate the content for this user control.
        Returns a `UIContent` instance.
```
get_invalidate_events() → Iterable[Event[object]]
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() → Optional[KeyBindingsBase]
The key bindings that are specific for this user control.
Return a KeyBindings object if some key bindings are specified, or None otherwise.

is_focussable() → bool
Tell whether this user control is focussable.

mouse_handler(mouse_event: prompt_toolkit.mouse_events.MouseEvent) → Not ImplementedOrNone
Handle mouse events.
When Not Implemented 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() → None
Request to move the cursor down. This happens when scrolling down and the cursor is completely at the top.

move_cursor_up() → None
Request to move the cursor up.

class prompt_toolkit.layout.UIContent(get_line: Callable[[int], List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]], line_count: int = 0, cursor_position: Optional[prompt_toolkit.data_structures.Point] = None, menu_position: Optional[prompt_toolkit.data_structures.Point] = None, show_cursor: bool = 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: int, width: int, get_line_prefix: Optional[Callable[[int, int], Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]], Callable[[Any]], slice_stop: Optional[int] = None] → int
Return the height that a given line would need if it is rendered in a space with the given width (using line wrapping).

Parameters
• get_line_prefix – None or a Window.get_line_prefix callable that returns the prefix to be inserted before this line.
• slice_stop – Wrap only “line[:slice_stop]” and return that partial result. This is needed for scrolling the window correctly when line wrapping.
Returns The computed height.

Other

Sizing

class prompt_toolkit.layout.Dimension(min: Optional[int] = None, max: Optional[int] = None, weight: Optional[int] = None, preferred: Optional[int] = 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 with 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: int) → prompt_toolkit.layout.dimension.Dimension

Return a Dimension with an exact size. (min, max and preferred set to amount).

is_zero() → bool

True if this Dimension represents a zero size.

classmethod zero() → prompt_toolkit.layout.dimension.Dimension

Create a dimension that represents a zero size. (Used for ‘invisible’ controls.)

Margins

class prompt_toolkit.layout.Margin

Base interface for a margin.

abstract create_margin(window_render_info: WindowRenderInfo, width: int, height: int) →
List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]

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.)

abstract get_width(get_ui_content: Callable[[], prompt_toolkit.layout.controls.UIContent]) → int

Return the width that this margin is going to consume.

Parameters **get_ui_content** – Callable that asks the user control to create a UIControl instance. This can be used for instance to obtain the number of lines.

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.

create_margin(window_render_info: WindowRenderInfo, width: int, height: int) → List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]

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: Callable[[], prompt_toolkit.layout.controls.UIContent]) → int

Return the width that this margin is going to consume.

Parameters get_ui_content – Callable that asks the user control to create a UIControl instance. This can be used for instance to obtain the number of lines.


Margin displaying a scrollbar.

Parameters display_arrows – Display scroll up/down arrows.

create_margin(window_render_info: WindowRenderInfo, width: int, height: int) → List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]

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: Callable[[], prompt_toolkit.layout.controls.UIContent]) → int

Return the width that this margin is going to consume.

Parameters get_ui_content – Callable that asks the user control to create a UIControl instance. This can be used for instance to obtain the number of lines.


Wrapper around other Margin classes to show/hide them.

create_margin(window_render_info: WindowRenderInfo, width: int, height: int) → List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]

Creates a margin. This should return a list of (style_str, text) tuples.

3.11. Reference
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.)

```python
def get_width(get_ui_content: Callable[[], prompt_toolkit.layout.controls.UIContent]) -> int
```

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.

```python
class prompt_toolkit.layout.PromptMargin(get_prompt: Callable[[], List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]]], get_continuation: Optional[Callable[[int, int, bool], List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]]] = None)
```

[Deprecated]

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.

This `PromptMargin` implementation has been largely superseded in favor of the `get_line_prefix` attribute of Window. The reason is that a margin is always a fixed width, while `get_line_prefix` can return a variable width prefix in front of every line, making it more powerful, especially for line continuations.

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.

```python
def create_margin(window_render_info: WindowRenderInfo, width: int, height: int) -> List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]]
```

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.)

```python
def get_width(get_ui_content: Callable[[], prompt_toolkit.layout.controls.UIContent]) -> int
```

Width to report to the Window.
Completion Menus

class prompt_toolkit.layout.CompletionsMenu
(max_height: Optional[int] = None, scroll_offset:
Union[int, Callable[[], int]] = 0, extra_filter:
Union[prompt_toolkit.filters.base.Filter, bool] = True,

class prompt_toolkit.layout.MultiColumnCompletionsMenu
(min_rows: int = 3,
suggested_max_column_width: int = 30,
show_meta:
Union[prompt_toolkit.filters.base.Filter, bool] = True, extra_filter:
Union[prompt_toolkit.filters.base.Filter, bool] = True, z_index: int = 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

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.AfterInput
(text: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]], Callable[[], Any]], style: str = '')

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.

apply_transformation(ti: prompt_toolkit.layout.processors.TransformationInput) →
prompt_toolkit.layout.processors.Transformation

Apply transformation. Returns a Transformation instance.

Parameters transformation_input – TransformationInput object.

class prompt_toolkit.layout.processors.AppendAutoSuggestion
(style: str = 'class:auto-suggestion')

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

apply_transformation(ti: prompt_toolkit.layout.processors.TransformationInput) →
prompt_toolkit.layout.processors.Transformation

Apply transformation. Returns a Transformation instance.

Parameters transformation_input – TransformationInput object.

class prompt_toolkit.layout.processors.BeforeInput
(text: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]]], Callable[[], Any]], style: str = '')

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.

**apply_transformation**

```
apply_transformation(ti: prompt_toolkit.layout.processors.TransformationInput) ->
prompt_toolkit.layout.processors.Transformation
```

Apply transformation. Returns a Transformation instance.

**Parameters**

transformation_input – TransformationInput object.

**class**

```
class prompt_toolkit.layout.processors.ConditionalProcessor(processor:
prompt_toolkit.layout.processors.Processor, filter:
Union[prompt_toolkit.filters.base.Filter, bool])
```

Processor that applies another processor, according to a certain condition. Example:

```python
# 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=[
    ConditionalProcessor(HighlightSearchProcessor(),
    Condition(highlight_enabled))])
```

**Parameters**

• processor – Processor instance.

• filter – Filter instance.

**apply_transformation**

```
apply_transformation(transformation_input: prompt_toolkit.layout.processors.TransformationInput) ->
prompt_toolkit.layout.processors.Transformation
```

Apply transformation. Returns a Transformation instance.

**Parameters**

transformation_input – TransformationInput object.

**class**

```
class prompt_toolkit.layout.processors.DisplayMultipleCursors
```

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

**apply_transformation**

```
apply_transformation(transformation_input: prompt_toolkit.layout.processors.TransformationInput) ->
prompt_toolkit.layout.processors.Transformation
```

Apply transformation. Returns a Transformation instance.

**Parameters**

transformation_input – TransformationInput object.

**class**

```
class prompt_toolkit.layout.processors.DummyProcessor
```

A Processor that doesn’t do anything.

**apply_transformation**

```
apply_transformation(transformation_input: prompt_toolkit.layout.processors.TransformationInput) ->
prompt_toolkit.layout.processors.Transformation
```

Apply transformation. Returns a Transformation instance.

**Parameters**

transformation_input – TransformationInput object.

**class**

```
```

Processor class that dynamically returns any Processor.
Parameters `get_processor` – Callable that returns a `Processor` instance.

```python
def apply_transformation(ti: TransformationInput) -> Transformation:
    Apply transformation. Returns a `Transformation` instance.
```

Parameters `transformation_input` – `TransformationInput` object.

class 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.

```python
class HighlightMatchingBracketProcessor(chars: str = '/[]()<>', max_cursor_distance: int = 1000):
    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.)

    ```python
def apply_transformation(transformation_input: TransformationInput) -> Transformation:
    Apply transformation. Returns a `Transformation` instance.
```

Parameters `transformation_input` – `TransformationInput` object.

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

```python
def apply_transformation(transformation_input: TransformationInput) -> Transformation:
    Apply transformation. Returns a `Transformation` instance.
```

Parameters `transformation_input` – `TransformationInput` object.

class HighlightSelectionProcessor
Processor that highlights the selection in the document.

```python
def apply_transformation(transformation_input: TransformationInput) -> Transformation:
    Apply transformation. Returns a `Transformation` instance.
```

Parameters `transformation_input` – `TransformationInput` object.

class PasswordProcessor
Processor that masks the input. (For passwords.)

```python
def apply_transformation(ti: TransformationInput) -> Transformation:
    Apply transformation. Returns a `Transformation` instance.
```

Parameters `transformation_input` – `TransformationInput` object.
class prompt_toolkit.layout.processors.Processor
    Manipulate the fragments for a given line in a BufferControl.

    abstract apply_transformation(transformation_input:
                                    prompt_toolkit.layout.processors.TransformationInput) →
                                    prompt_toolkit.layout.processors.Transformation
    Apply transformation. Returns a Transformation instance.

    Parameters transformation_input – TransformationInput object.

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.

    apply_transformation(ti: prompt_toolkit.layout.processors.TransformationInput) →
                              prompt_toolkit.layout.processors.Transformation
    Apply transformation. Returns a Transformation instance.

    Parameters transformation_input – TransformationInput object.

class prompt_toolkit.layout.processors.ShowArg
    Display the ‘arg’ in front of the input.

    This was used by the PromptSession, but now it uses the Window.get_line_prefix function instead.

class prompt_toolkit.layout.processors.ShowLeadingWhiteSpaceProcessor(get_char:
                                                                      Optional[Callable[[],
                                                                      str]] = None,
                                                                      style: str = '/quotesingle.ts1' |
                                                                      class:leading-whitespace)
    Make leading whitespace visible.

    Parameters get_char – Callable that returns one character.

    apply_transformation(ti: prompt_toolkit.layout.processors.TransformationInput) →
                                   prompt_toolkit.layout.processors.Transformation
    Apply transformation. Returns a Transformation instance.

    Parameters transformation_input – TransformationInput object.

class prompt_toolkit.layout.processors.ShowTrailingWhiteSpaceProcessor(get_char:
                                                                       Optional[Callable[[],
                                                                       str]] = None,
                                                                       style: str = '/quotesingle.ts1' |
                                                                       class:training-whitespace)
    Make trailing whitespace visible.

    Parameters get_char – Callable that returns one character.

    apply_transformation(ti: prompt_toolkit.layout.processors.TransformationInput) →
                                   prompt_toolkit.layout.processors.Transformation
    Apply transformation. Returns a Transformation instance.

    Parameters transformation_input – TransformationInput object.

class prompt_toolkit.layout.processors.TabsProcessor(tabstop: Union[int, Callable[[], int]] = 4,
                                                      char1: Union[str, Callable[[], str]] = '|
                                                      char2: Union[str, Callable[[], str]] = ' ', style: str = 'class:tab')
    Render tabs as spaces (instead of ^I) or make them visible (for instance, by replacing them with dots.)

166 Chapter 3. Table of contents
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.

```python
```

Apply transformation. Returns a **Transformation** instance.

Parameters **transformation_input** – **TransformationInput** object.

```python
class prompt_toolkit.layout.processors.Transformation(fragments: List[Union[Tuple[str, str], Tuple[str, str, Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], source_to_display: Optional[Callable[[int], int]] = None, display_to_source: Optional[Callable[[int], int]] = 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.

```python
```

Parameters

- **buffer_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.)

```python
```

Merge multiple **Processor** objects into one.

3.11. Reference
Utils


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.

Screen

class prompt_toolkit.layout.screen.Char(char: str = r'', style: str = r'')

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.)

class prompt_toolkit.layout.screen.Screen(default_char: Optional[prompt_toolkit.layout.screen.Char] = None, initial_width: int = 0, initial_height: int = 0)

Two dimensional buffer of Char instances.

append_style_to_content(style_str: str) → None

For all the characters in the screen. Set the style string to the given style_str.

cursor_positions: Dict[Window, prompt_toolkit.data_structures.Point]

Position of the cursor.

draw_all_floats() → None

Draw all float functions in order of z-index.

draw_with_z_index(z_index: int, draw_func: Callable[[], None]) → None

Add a draw-function for a Window which has a >= 0 z_index. This will be postponed until draw_all_floats is called.


Fill the content of this area, using the given style. The style is prepended before whatever was here before.

get_cursor_position(window: Window) → prompt_toolkit.data_structures.Point

Get the cursor position for a given window. Returns a Point.

get_menu_position(window: Window) → prompt_toolkit.data_structures.Point

Get the menu position for a given window. (This falls back to the cursor position if no menu position was set.)

menu_positions: Dict[Window, prompt_toolkit.data_structures.Point]

(Optional) Where to position the menu. E.g. at the start of a completion. (We can’t use the cursor position, because we don’t want the completion menu to change its position when we browse through all the completions.)

set_cursor_position(window: Window, position: prompt_toolkit.data_structures.Point) → None

Set the cursor position for a given window.
**set_menu_position** *(window: Window, position: prompt_toolkit.data_structures.Point) → None*

Set the cursor position for a given window.

**show_cursor**

Visibility of the cursor.

**width**

Currently used width/height of the screen. This will increase when data is written to the screen.

**zero_width_escapes**: *DefaultDict[int, DefaultDict[int, str]]*

Escape sequences to be injected.

### 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.


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.Button(text: str, handler: Optional[Callable[[], None]] = None, width: int = 12, left_symbol: str = '<', right_symbol: str = '>')

Clickable button.

Parameters

- **text** – The caption for the button.
- **handler** – `None` or callable. Called when the button is clicked. No parameters are passed to this callable. Use for instance Python's `functools.partial` to pass parameters to this callable if needed.
- **width** – Width of the button.

class prompt_toolkit.widgets.Checkbox(text: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], Callable[[], Any]]] = None), checked: bool = False)

Backward compatibility util: creates a 1-sized CheckboxList

Parameters **text** – the text
class prompt_toolkit.widgets.FormattedTextToolbar(text: Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], Callable[[], Any]]], style: str = '', **kw: Any)


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.HorizontalLine

A simple horizontal line with a height of 1.

Chapter 3. Table of contents
class prompt_toolkit.widgets.Label

Widget that displays the given text. It is not editable or focusable.

Parameters
- **text** – Text to display. Can be multiline. All value types accepted by prompt_toolkit.layout.FormattedTextControl are allowed, including a callable.
- **style** – A style string.
- **width** – When given, use this width, rather than calculating it from the text size.
- **dont_extend_width** – When True, don’t take up more width than preferred, i.e. the length of the longest line of the text, or value of width parameter, if given. True by default
- **dont_extend_height** – When True, don’t take up more width than the preferred height, i.e. the number of lines of the text. False by default.

class prompt_toolkit.widgets.MenuContainer

Parameters
- **floats** – List of extra Float objects to display.
- **menu_items** – List of MenuItem objects.

class prompt_toolkit.widgets.Radiolist
(values: Sequence[Tuple[prompt_toolkit.widgets.base._T, Optional[Union[str, MagicFormattedText, List[Union[Tuple[str, str], Tuple[str, str], Callable[[prompt_toolkit.mouse_events.MouseEvent], None]]], Callable[[Any]], Any]]]], default: Optional[prompt_toolkit.widgets.base._T] = None)

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:

Parameters

- **vi_mode** – Display ‘/’ and ‘?’ instead of I-search.
- **ignore_case** – Search case insensitive.

class prompt_toolkit.widgets.Shadow:

Parameters **body** – Another container object.

class prompt_toolkit.widgets.SystemToolbar:

Parameters **prompt** – Prompt to be displayed to the user.
A simple input field.

This is a higher level abstraction on top of several other classes with sane defaults.

This widget does have the most common 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`.

Buffer attributes:

**Parameters**

- `text` – The initial text.
- `multiline` – If True, allow multiline input.
- `completer` – `Completer` instance for auto completion.
- `complete_while_typing` – Boolean.
- `accept_handler` – Called when `Enter` is pressed (This should be a callable that takes a buffer as input).
• **history** – *History* instance.
• **auto_suggest** – *AutoSuggest* instance for input suggestions.

BufferControl attributes:

**Parameters**

• **password** – When *True*, display using asterisks.
• **focusable** – When *True*, allow this widget to receive the focus.
• **focus_on_click** – When *True*, focus after mouse click.
• **input_processors** – *None* or a list of *Processor* objects.
• **validator** – *None* or a *Validator* object.

Window attributes:

**Parameters**

• **lexer** – *Lexer* instance for syntax highlighting.
• **wrap_lines** – When *True*, don’t scroll horizontally, but wrap lines.
• **width** – Window width. (*Dimension* object.)
• **height** – Window height. (*Dimension* object.)
• **scrollbar** – When *True*, display a scroll bar.
• **style** – A style string.
• **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.
• **get_line_prefix** – None or a callable that returns formatted text to be inserted before a line. It takes a line number (int) and a wrap_count and returns formatted text. This can be used for implementation of line continuations, things like Vim “breakindent” and so on.

Other attributes:

**Parameters**

• **search_field** – An optional *SearchToolbar* object.

**property accept_handler**: `Optional[Callable[[prompt_toolkit.buffer.Buffer], bool]]`

The accept handler. Called when the user accepts the input.

**property document**: `prompt_toolkit.document.Document`

The *Buffer* document (text + cursor position).

**property text**: `str`

The *Buffer* text.

class `prompt_toolkit.widgets.VerticalLine`

A simple vertical line with a width of 1.
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, it’s 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:

```python
filter = has_focus('default') & ~ has_selection
```

class prompt_toolkit.filters.Always
Always enable feature.

class prompt_toolkit.filters.Condition(func: Callable[[], bool])
Turn any callable into a Filter. The callable is supposed to not take any arguments.

This can be used as a decorator:

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

Parameters

- **func** – Callable which takes no inputs and returns a boolean.

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.

prompt_toolkit.filters.HasFocus(value: FocusableElement) → prompt_toolkit.filters.base.Condition
Enable when this buffer has the focus.

Check whether a given editing mode is active. (Vi or Emacs.)

class prompt_toolkit.filters.Never
Never enable feature.

prompt_toolkit.filters.has_focus(value: FocusableElement) → prompt_toolkit.filters.base.Condition
Enable when this buffer has the focus.

prompt_toolkit.filters.in_editing_mode(editing_mode: prompt_toolkit.enums.EditingMode) → prompt_toolkit.filters.base.Condition
Check whether a given editing mode is active. (Vi or Emacs.)

prompt_toolkit.filters.is_true(value: Union[prompt_toolkit.filters.base.Filter, bool]) → bool
Test whether value is True. In case of a Filter, call it.

Parameters

- **value** – Boolean or Filter instance.

prompt_toolkit.filters.to_filter(bool_or_filter: Union[prompt_toolkit.filters.base.Filter, bool]) → prompt_toolkit.filters.base.Filter
Accept both booleans and Filters as input and turn it into a Filter.
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: Callable[[], bool])
    Turn any callable into a Filter. The callable is supposed to not take any arguments.
    
    This can be used as a decorator:

    ```python
    @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.is_true(value: Union[prompt_toolkit.filters.base.Filter, bool]) → bool
    Test whether value is True. In case of a Filter, call it.
    
    Parameters
    ----------
    value -- Boolean or Filter instance.

prompt_toolkit.filters.utils.to_filter(bool_or_filter: Union[prompt_toolkit.filters.base.Filter, bool]) → prompt_toolkit.filters.base.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(value: FocusableElement) → prompt_toolkit.filters.base.Condition
    Enable when this buffer has the focus.

prompt_toolkit.filters.app.in_editing_mode(editing_mode: prompt_toolkit.enums.EditingMode) → prompt_toolkit.filters.base.Condition
    Check whether a given editing mode is active. (Vi or Emacs.)

### 3.11.20 Key binding

    Wraps around a KeyBindings. Disable/enable all the key bindings according to the given (additional) filter:

    ```python
    @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/disable according to the given filter.

    Parameters
    ----------
    • registries -- List of KeyBindings objects.
    • filter -- Filter object.
class prompt_toolkit.key_binding.DynamicKeyBindings(get_key_bindings: Callable[[], Optional[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase]]):
    KeyBindings class that can dynamically returns any KeyBindings.

    Parameters
    get_key_bindings – Callable that returns a KeyBindings instance.

class prompt_toolkit.key_binding.KeyBindings:
    A container for a set of key bindings.

Example usage:

```python
def prompt_toolkit:
    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.
```


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.)


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.)

**property bindings**: List[**prompt_toolkit.key_binding.key_bindings.Binding**]

List of *Binding* objects. (These need to be exposed, so that *KeyBindings* objects can be merged together.)

**get_bindings_for_keys** *(keys: Tuple[Union[prompt_toolkit.keys.Keys, str], ...])* → List[prompt_toolkit.key_binding.key_bindings.Binding]  

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: Tuple[Union[prompt_toolkit.keys.Keys, str], ...])* → List[prompt_toolkit.key_binding.key_bindings.Binding]  

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: Union[prompt_toolkit.keys.Keys, str, Callable[[KeyPressEvent], Union[NotImplementedOrNone, Awaitable[NotImplementedOrNone]]]])* → None  

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**:

```py
remove(handler)  # Pass handler.
remove('c-x', 'c-a')  # Or pass the key bindings.
```

**remove_binding** *(args: Union[prompt_toolkit.keys.Keys, str, Callable[[KeyPressEvent], Union[NotImplementedOrNone, Awaitable[NotImplementedOrNone]]]])* → None  

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**:

```py
remove(handler)  # Pass handler.
remove('c-x', 'c-a')  # Or pass the key bindings.
```

class **prompt_toolkit.key_binding.KeyBindingsBase**

Interface for a *KeyBindings*.  

178 Chapter 3. Table of contents
abstract property bindings: List[prompt_toolkit.key_binding.key_bindings.Binding]
List of Binding objects. (These need to be exposed, so that KeyBindings objects can be merged together.)

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.

abstract get_bindings_starting_with_keys(keys: Tuple[Union[prompt_toolkit.keys.Keys, str], ...])
→ List[prompt_toolkit.key_binding.key_bindings.Binding]
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.

prompt_toolkit.keybinding.merge_key_bindings(bindings: Sequence[prompt_toolkit.key_binding.key_bindings.KeyBindingsBase]) → prompt_toolkit.key_binding.key_bindings._MergedKeyBindings
Merge multiple Keybinding objects together.

Usage:

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

Default key bindings:

```
key_bindings = load_key_bindings()
app = Application(key_bindings=key_bindings)
```

prompt_toolkit.keybinding.defaults.load_key_bindings() → prompt_toolkit.key_binding.key_bindings.KeyBindingsBase
Create a KeyBindings object that contains the default key bindings.

class prompt_toolkit.key_binding.vi_state.InputMode(value)
An enumeration.

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

property input_mode: prompt_toolkit.key_binding.vi_state.InputMode
Get InputMode.

last_character_find: Optional[prompt_toolkit.key_binding.vi_state.CharacterFind]
None or CharacterFind instance. (This is used to repeat the last search in Vi mode, by pressing the ‘n’ or ‘N’ in navigation mode.)

named_registers: Dict[str, prompt_toolkit.clipboard.base.ClipboardData]
Named registers. Maps register name (e.g. ‘a’) to ClipboardData instances.

reset() → None
Reset state, go back to the given mode. INSERT by default.

tilde_operator
When true, make ~ act as an operator.
waiting_for_digraph
Waiting for digraph.

An `KeyProcessor` receives callbacks for the keystrokes parsed from the input in the `InputStream` instance.

The `KeyProcessor` will according to the implemented keybindings call the correct callbacks when new key presses are feed through `feed`.

```python
class prompt_toolkit.key_binding.key_processor.KeyPress(key: Union[prompt_toolkit.keys.Keys, str], data: Optional[str] = None)
```

**Parameters**

- **key** – A `Keys` instance or text (one character).
- **data** – The received string on stdin. (Often vt100 escape codes.)

```python
class prompt_toolkit.key_binding.key_processor.KeyPressEvent(key_processor_ref: weakref.ReferenceType[KeyProcessor], arg: Optional[str], key_sequence: List[prompt_toolkit.key_binding.key_processor.KeyPress], previous_key_sequence: List[prompt_toolkit.key_binding.key_processor.KeyPress], is_repeat: bool)
```

Key press event, delivered to key bindings.

**Parameters**

- **key_processor_ref** – Weak reference to the `KeyProcessor`.
- **arg** – Repetition argument.
- **key_sequence** – List of `KeyPress` instances.
- **previous_key_sequence** – Previous list of `KeyPress` instances.
- **is_repeat** – True when the previous event was delivered to the same handler.

```python
property app: Application[Any]
    The current `Application` object.

append_to_arg_count(data: str) → None
    Add digit to the input argument.

    Parameters
    data – the typed digit as string
```

```python
property arg: int
    Repetition argument.

property arg_present: bool
    True if repetition argument was explicitly provided.

property cli: Application[Any]
    For backward-compatibility.

property current_buffer: Buffer
    The current buffer.

is_repeat
    True when the previous key sequence was handled by the same handler.
class prompt_toolkit.key_binding.key_processor.KeyProcessor(key_bindings):
    Statemachine that receives KeyPress instances and according to the key bindings in the given KeyBindings, calls the matching handlers.

    p = KeyProcessor(key_bindings)
    # Send keys into the processor.
    p.feed(KeyPress(Keys.ControlX, ''))
    p.feed(KeyPress(Keys.ControlC, ''))
    # Process all the keys in the queue.
    p.process_keys()
    # Now the ControlX-ControlC callback will be called if this sequence is
    # registered in the key bindings.

Parameters key_bindings – KeyBindingsBase instance.

empty_queue() → List[prompt_toolkit.key_binding.key_processor.KeyPress]
    Empty the input queue. Return the unprocessed input.

feed(key_press: prompt_toolkit.key_binding.key_processor.KeyPress, first: bool = False) → None
    Add a new KeyPress to the input queue. (Don’t forget to call process_keys in order to process the queue.)

    Parameters first – If true, insert before everything else.

feed_multiple(key_presses: List[prompt_toolkit.key_binding.key_processor.KeyPress], first: bool = False) → None

    Parameters first – If true, insert before everything else.

process_keys() → None
    Process all the keys in the input_queue. (To be called after feed.)

    Note: because of the feed/process_keys separation, it is possible to call feed from inside a key binding.
    This function keeps looping until the queue is empty.

send_sigint() → None
    Send SIGINT. Immediately call the SIGINT key handler.

3.11.21 Eventloop

prompt_toolkit.eventloop.get_event_loop() → asyncio.events.AbstractEventLoop
    Backward compatible way to get the event loop

prompt_toolkit.eventloop.get_traceback_from_context(context: Dict[str, Any]) → Optional[types.TracebackType]
    Get the traceback object from the context.
### 3.11.22 Input

**class prompt_toolkit.input.DummyInput**
Input for use in a *DummyApplication*

```python
attach(input_ready_callback: Callable[[], None]) → AbstractContextManager[None]
Return a context manager that makes this input active in the current event loop.
```

**property closed: bool**
Should be true when the input stream is closed.

```python
cooked_mode() → AbstractContextManager[None]
Context manager that turns the input into cooked mode.
```

```python
detach() → AbstractContextManager[None]
Return a context manager that makes sure that this input is not active in the current event loop.
```

```python
fileno() → int
Fileno for putting this in an event loop.
```

```python
raw_mode() → AbstractContextManager[None]
Context manager that turns the input into raw mode.
```

```python
read_keys() → List[prompt_toolkit.key_binding.key_processor.KeyPress]
Return a list of Key objects which are read/parsed from the input.
```

```python
typeahead_hash() → str
Identifier for storing type ahead key presses.
```

**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.

```python
abstract attach(input_ready_callback: Callable[[], None]) → AbstractContextManager[None]
Return a context manager that makes this input active in the current event loop.
```

```python
close() → None
Close input.
```

**abstract property closed: bool**
Should be true when the input stream is closed.

```python
abstract cooked_mode() → AbstractContextManager[None]
Context manager that turns the input into cooked mode.
```

```python
abstract detach() → AbstractContextManager[None]
Return a context manager that makes sure that this input is not active in the current event loop.
```

```python
abstract fileno() → int
Fileno for putting this in an event loop.
```

```python
flush() → None
The event loop can call this when the input has to be flushed.
```

```python
flush_keys() → List[prompt_toolkit.key_binding.key_processor.KeyPress]
Flush the underlying parser. and return the pending keys. (Used for vt100 input.)
```

```python
abstract raw_mode() → AbstractContextManager[None]
Context manager that turns the input into raw mode.
```

```python
abstract read_keys() → List[prompt_toolkit.key_binding.key_processor.KeyPress]
Return a list of Key objects which are read/parsed from the input.
```
abstract typeahead_hash() → str
    Identifier for storing type ahead key presses.

prompt_toolkit.input.create_input(stdin: Optional[TextIO] = None, always_prefer_tty: bool = False) →
prompt_toolkit.input.base.Input
    Create the appropriate Input object for the current os/environment.

Parameters always_prefer_tty – When set, if sys.stdin is connected to a Unix pipe, check whether sys.stdout or sys.stderr are connected to a pseudo terminal. If so, open the tty for reading instead of reading for sys.stdin. (We can open stdout or stderr for reading, this is how a $PAGER works.)

prompt_toolkit.input.create_pipe_input() → prompt_toolkit.input.base.PipeInput
    Create an input pipe. This is mostly useful for unit testing.

class prompt_toolkit.input.vt100.Vt100Input(stdin: TextIO)
    Vt100 input for Posix systems. (This uses a posix file descriptor that can be registered in the event loop.)

    attach(input_ready_callback: Callable[[], None]) → AbstractContextManager[None]
        Return a context manager that makes this input active in the current event loop.

property closed: bool
    Should be true when the input stream is closed.

cooked_mode() → AbstractContextManager[None]
    Context manager that turns the input into cooked mode.

detach() → AbstractContextManager[None]
    Return a context manager that makes sure that this input is not active in the current event loop.

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

flush_keys() → List[prompt_toolkit.key_binding.key_processor.KeyPress]
    Flush pending keys and return them. (Used for flushing the ‘escape’ key.)

raw_mode() → AbstractContextManager[None]
    Context manager that turns the input into raw mode.

read_keys() → List[prompt_toolkit.key_binding.key_processor.KeyPress]
    Read list of KeyPress.

typeahead_hash() → str
    Identifier for storing type ahead keys.

class prompt_toolkit.input.vt100.cooked_mode(fileno: int)
    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. ""

class prompt_toolkit.input.vt100.raw_mode(fileno: int)

    with raw_mode(stdin):
        """ the pseudo-terminal stdin is now used in raw mode ""

We ignore errors when executing tcgetattr fails.

Parser for VT100 input stream.
class prompt_toolkit.input.vt100Parser(feed_key_callback: Callable[[prompt_toolkit.key_binding.key_processor.KeyPress], None])

Parser for VT100 input stream. Data can be fed through the feed method and the given callback will be called with KeyPress objects.

```python
def callback(key):
    pass

i = Vt100Parser(callback)
i.feed('data{...}')
```

Attr **feed_key_callback** Function that will be called when a key is parsed.

**feed**(data: str) → None
Feed the input stream.

**feed_and_flush**(data: str) → None
Wrapper around feed and flush.

**flush**() → None
Flush the buffer of the input stream.

This will allow us to handle the escape key (or maybe meta) sooner. The input received by the escape key is actually the same as the first characters of e.g. Arrow-Up, so without knowing what follows the escape sequence, we don’t know whether escape has been pressed, or whether it’s something else. This flush function should be called after a timeout, and processes everything that’s still in the buffer as-is, so without assuming any characters will follow.

Mappings from VT100 (ANSI) escape sequences to the corresponding prompt_toolkit keys.

We are not using the terminfo/termcap databases to detect the ANSI escape sequences for the input. Instead, we recognize 99% of the most common sequences. This works well, because in practice, every modern terminal is mostly Xterm compatible.

Some useful docs: - Mintty: https://github.com/mintty/mintty/blob/master/wiki/Keycodes.md

### 3.11.23 Output

class prompt_toolkit.output.ColorDepth(value)
Possible color depth values for the output.

**DEPTH_1_BIT** = 'DEPTH_1_BIT'
One color only.

**DEPTH_24_BIT** = 'DEPTH_24_BIT'
24 bit True color.

**DEPTH_4_BIT** = 'DEPTH_4_BIT'
ANSI Colors.

**DEPTH_8_BIT** = 'DEPTH_8_BIT'
The default.

classmethod **default**() → prompt_toolkit.output.color_depth.ColorDepth
Return the default color depth for the default output.
classmethod from_env() \rightarrow Optional[prompt_toolkit.output.color_depth.ColorDepth]

Return the color depth if the $PROMPT_TOOLKIT_COLOR_DEPTH environment variable has been set.

This is a way to enforce a certain color depth in all prompt_toolkit applications.

class prompt_toolkit.output.DummyOutput

For testing. An output class that doesn’t render anything.

ask_for_cpr() \rightarrow None

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

bell() \rightarrow None

Sound bell.

clear_title() \rightarrow None

Clear title again. (or restore previous title.)

cursor_backward(amount: int) \rightarrow None

Move cursor amount place backward.

cursor_down(amount: int) \rightarrow None

Move cursor amount place down.

cursor_forward(amount: int) \rightarrow None

Move cursor amount place forward.

cursor_goto(row: int = 0, column: int = 0) \rightarrow None

Move cursor position.

cursor_up(amount: int) \rightarrow None

Move cursor amount place up.

disable_autowrap() \rightarrow None

Disable auto line wrapping.

disable_bracketed_paste() \rightarrow None

For vt100 only.

disable_mouse_support() \rightarrow None

Disable mouse.

enable_autowrap() \rightarrow None

Enable auto line wrapping.

enable_bracketed_paste() \rightarrow None

For vt100 only.

enable_mouse_support() \rightarrow None

Enable mouse.

encoding() \rightarrow str

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() \rightarrow None

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

erase_down() \rightarrow None

Erases the screen from the current line down to the bottom of the screen.

erase_end_of_line() \rightarrow None

Erases from the current cursor position to the end of the current line.
erase_screen() → None
   Erases the screen with the background colour and moves the cursor to home.

fileno() → int
   There is no sensible default for fileno().

flush() → None
   Write to output stream and flush.

get_default_color_depth() → prompt_toolkit.output.color_depth.ColorDepth
   Get default color depth for this output.

   This value will be used if no color depth was explicitly passed to the Application.

   Note: If the $PROMPT_TOOLKIT_COLOR_DEPTH environment variable has been set, then outputs.defaults.create_output will pass this value to the implementation as the default_color_depth, which is returned here. (This is not used when the output corresponds to a prompt_toolkit SSH/Telnet session.)

get_rows_below_cursor_position() → int
   For Windows only.

get_size() → prompt_toolkit.data_structures.Size
   Return the size of the output window.

hide_cursor() → None
   Hide cursor.

quit_alternate_screen() → None
   Leave the alternate screen buffer.

reset_attributes() → None
   Reset color and styling attributes.

reset_cursor_shape() → None
   Reset cursor shape.

scroll_buffer_to_prompt() → None
   For Win32 only.

   Set new color and styling attributes.

set_cursor_shape(cursor_shape: prompt_toolkit.cursor_shapes.CursorShape) → None
   Set cursor shape to block, beam or underline.

set_title(title: str) → None
   Set terminal title.

show_cursor() → None
   Show cursor.

write(data: str) → None
   Write text (Terminal escape sequences will be removed/escaped.)

write_raw(data: str) → None
   Write text.

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

   Actual implementations are Vt100_Output and Win32Output.
ask_for_cpr() → None
Asks for a cursor position report (CPR). (VT100 only.)

bell() → None
Sound bell.

abstract clear_title() → None
Clear title again. (or restore previous title.)

abstract cursor_backward(amount: int) → None
Move cursor amount place backward.

abstract cursor_down(amount: int) → None
Move cursor amount place down.

abstract cursor_forward(amount: int) → None
Move cursor amount place forward.

abstract cursor_goto(row: int = 0, column: int = 0) → None
Move cursor position.

abstract cursor_up(amount: int) → None
Move cursor amount place up.

abstract disable_autowrap() → None
Disable auto line wrapping.

disable_bracketedPaste() → None
For vt100 only.

abstract disable_mouse_support() → None
Disable mouse.

abstract enable_autowrap() → None
Enable auto line wrapping.

enable_bracketedPaste() → None
For vt100 only.

abstract enable_mouse_support() → None
Enable mouse.

abstract encoding() → str
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.)

abstract enter_alternate_screen() → None
Go to the alternate screen buffer. (For full screen applications).

abstract erase_down() → None
Erases the screen from the current line down to the bottom of the screen.

abstract erase_end_of_line() → None
Erases from the current cursor position to the end of the current line.

abstract erase_screen() → None
Erases the screen with the background colour and moves the cursor to home.

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

abstract flush() → None
Write to output stream and flush.

3.11. Reference
abstract \texttt{get_default_color_depth()} \rightarrow \texttt{prompt_toolkit.output.color_depth.ColorDepth}

Get default color depth for this output.

This value will be used if no color depth was explicitly passed to the \emph{Application}.

\textbf{Note:} If the \$PROMPT_TOOLKIT\_COLOR\_DEPTH\$ environment variable has been set, then \texttt{outputs.defaults.create_output} will pass this value to the implementation as the default_color_depth, which is returned here. (This is not used when the output corresponds to a prompt_toolkit SSH/Telnet session.)

\texttt{get_rows_below_cursor_position()} \rightarrow \texttt{int}

For Windows only.

abstract \texttt{get_size()} \rightarrow \texttt{prompt_toolkit.data_structures.Size}

Return the size of the output window.

abstract \texttt{hide_cursor()} \rightarrow \texttt{None}

Hide cursor.

abstract \texttt{quit_alternate_screen()} \rightarrow \texttt{None}

Leave the alternate screen buffer.

abstract \texttt{reset_attributes()} \rightarrow \texttt{None}

Reset color and styling attributes.

\texttt{reset_cursor_key_mode()} \rightarrow \texttt{None}

For vt100 only. Put the terminal in normal cursor mode (instead of application mode).

See: https://vt100.net/docs/vt100-ug/chapter3.html

abstract \texttt{reset_cursor_shape()} \rightarrow \texttt{None}

Reset cursor shape.

\texttt{property responds_to_cpr: bool}

\emph{True} if the \emph{Application} can expect to receive a CPR response after calling \texttt{ask_for_cpr} (this will come back through the corresponding \emph{Input}).

This is used to determine the amount of available rows we have below the cursor position. In the first place, we have this so that the drop down autocompletion menus are sized according to the available space.

On Windows, we don’t need this, there we have \texttt{get_rows_below_cursor_position}.

\texttt{scroll_buffer_to_prompt()} \rightarrow \texttt{None}

For Win32 only.

abstract \texttt{set_attributes(\texttt{attrs: prompt_toolkit.styles.base.Attrs, color_depth: prompt_toolkit.output.color_depth.ColorDepth})} \rightarrow \texttt{None}

Set new color and styling attributes.

abstract \texttt{set_cursor_shape(\texttt{cursor_shape: prompt_toolkit.cursor_shapes.CursorShape})} \rightarrow \texttt{None}

Set cursor shape to block, beam or underline.

abstract \texttt{set_title(\texttt{title: str})} \rightarrow \texttt{None}

Set terminal title.

abstract \texttt{show_cursor()} \rightarrow \texttt{None}

Show cursor.

abstract \texttt{write(\texttt{data: str})} \rightarrow \texttt{None}

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

abstract \texttt{write_raw(\texttt{data: str})} \rightarrow \texttt{None}

Write text.
prompt_toolkit.output.create_output(stdout: Optional[TextIO] = None, always_prefer_tty: bool = False) → prompt_toolkit.output.base.Output

Return an Output instance for the command line.

Parameters

- **stdout** – The stdout object
- **always_prefer_tty** – When set, look for sys.stderr if sys.stdout is not a TTY. Useful if sys.stdout is redirected to a file, but we still want user input and output on the terminal.

By default, this is False. If sys.stdout is not a terminal (maybe it’s redirected to a file), then a PlaintextOutput will be returned. That way, tools like print_formatted_text will write plain text into that file.

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/


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() → None
    Asks for a cursor position report (CPR).

bell() → None
    Sound bell.

clear_title() → None
    Clear title again. (or restore previous title.)

cursor_backward(amount: int) → None
    Move cursor amount place backward.

cursor_down(amount: int) → None
    Move cursor amount place down.

cursor_forward(amount: int) → None
    Move cursor amount place forward.

cursor_goto(row: int = 0, column: int = 0) → None
    Move cursor position.

cursor_up(amount: int) → None
    Move cursor amount place up.
disable_autowrap() → None
   Disable auto line wrapping.

disable_bracketed_paste() → None
   For vt100 only.

disable_mouse_support() → None
   Disable mouse.

enable_autowrap() → None
   Enable auto line wrapping.

enable_bracketed_paste() → None
   For vt100 only.

enable_mouse_support() → None
   Enable mouse.

encoding() → str
   Return encoding used for stdout.

enter_alternate_screen() → None
   Go to the alternate screen buffer. (For full screen applications).

erase_down() → None
   Erases the screen from the current line down to the bottom of the screen.

erase_end_of_line() → None
   Erases from the current cursor position to the end of the current line.

erase_screen() → None
   Erases the screen with the background colour and moves the cursor to home.

fileno() → int
   Return file descriptor.

flush() → None
   Write to output stream and flush.

classmethod from_pty(stdout: TextIO, term: Optional[str] = None, default_color_depth:
   Optional[prompt_toolkit.output.color_depth.ColorDepth] = None, enable_bell:
   bool = True) → prompt_toolkit.output.vt100.Vt100_Output
   Create an Output class from a pseudo terminal. (This will take the dimensions by reading the pseudo
   terminal attributes.)

get_default_color_depth() → prompt_toolkit.output.color_depth.ColorDepth
   Return the default color depth for a vt100 terminal, according to the our term value.
   We prefer 256 colors almost always, because this is what most terminals support these days, and is a good
   default.

get_size() → prompt_toolkit.data_structures.Size
   Return the size of the output window.

hide_cursor() → None
   Hide cursor.

quit_alternate_screen() → None
   Leave the alternate screen buffer.

reset_attributes() → None
   Reset color and styling attributes.
**reset_cursor_key_mode**() → None
For vt100 only. Put the terminal in cursor mode (instead of application mode).

**reset_cursor_shape**() → None
Reset cursor shape.

**property** responds_to_cpr: bool
*True* if the Application can expect to receive a CPR response after calling ask_for_cpr (this will come back through the corresponding Input).

This is used to determine the amount of available rows we have below the cursor position. In the first place, we have this so that the drop down autocompletion menus are sized according to the available space.

On Windows, we don’t need this, there we have get_rows_below_cursor_position.

Create new style and output.

**Parameters**

*attrs* – Attrs instance.

**set_cursor_shape**(cursor_shape: prompt_toolkit.cursor_shapes.CursorShape) → None
Set cursor shape to block, beam or underline.

**set_title**(title: str) → None
Set terminal title.

**show_cursor**() → None
Show cursor.

**write**(data: str) → None
Write text to output. (Removes vt100 escape codes. – used for safely writing text.)

**write_raw**(data: str) → None
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:**

```python
with patch_stdout(application):
    ...
    application.run()
    ...
```

Multiple applications can run in the body of the context manager, one after the other.

**class** prompt_toolkit.patch_stdout.StdoutProxy(sleep_between_writes: float = 0.2, raw: bool = False)
File-like object, which prints everything written to it, output above the current application/prompt. This class is compatible with other file objects and can be used as a drop-in replacement for *sys.stdout* or can for instance be passed to logging.StreamHandler.

The current application, above which we print, is determined by looking what application currently runs in the AppSession that is active during the creation of this instance.
This class can be used as a context manager.

In order to avoid having to repaint the prompt continuously for every little write, a short delay of `sleep_between_writes` seconds will be added between writes in order to bundle many smaller writes in a short timespan.

**close() → None**

Stop `StdoutProxy` proxy.

This will terminate the write thread, make sure everything is flushed and wait for the write thread to finish.

**flush() → None**

Flush buffered output.

```
prompt_toolkit.patch_stdout.patch_stdout(raw: bool = False) → Generator[None, None, None]
```

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.

**Warning:** If a new event loop is installed using `asyncio.set_event_loop()`, then make sure that the context manager is applied after the event loop is changed. Printing to stdout will be scheduled in the event loop that’s active when the context manager is created.

**Parameters**

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

- genindex
- modindex
- search

*Prompt_toolkit was created by Jonathan Slenders.*
prompt_toolkit.application, 88
prompt_toolkit.auto_suggest, 137
prompt_toolkit.buffer, 98
prompt_toolkit.clipboard, 104
prompt_toolkit.clipboard.pyperclip, 106
prompt_toolkit.completion, 106
prompt_toolkit.document, 113
prompt_toolkit.enums, 117
prompt_toolkit.eventloop, 181
prompt_toolkit.filters, 175
prompt_toolkit.filters.app, 176
prompt_toolkit.filters.utils, 176
prompt_toolkit.formatted_text, 96
prompt_toolkit.history, 117
prompt_toolkit.input, 182
prompt_toolkit.input.ansi_escape_sequences, 184
prompt_toolkit.input.vt100, 183
prompt_toolkit.input.vt100_parser, 183
prompt_toolkit.key_binding, 176
prompt_toolkit.key_binding.defaults, 179
prompt_toolkit.key_binding.key_processor, 180
prompt_toolkit.key_binding.vi_state, 179
prompt_toolkit.keys, 119
prompt_toolkit.layout, 143
prompt_toolkit.layout.processors, 163
prompt_toolkit.layout.screen, 168
prompt_toolkit.layout.utils, 168
prompt_toolkit.lexers, 141
prompt_toolkit.output, 184
prompt_toolkit.output.vt100, 189
prompt_toolkit.patch_stdout, 191
prompt_toolkit.renderer, 140
prompt_toolkit.selection, 104
prompt_toolkit.shortcuts, 124
prompt_toolkit.shortcuts.progress_bar.formatters, 135
prompt_toolkit.styles, 119
prompt_toolkit.validation, 136
prompt_toolkit.widgets, 169
A

accept_handler  (prompt_toolkit.widgets.TextArea property), 174
add()  (prompt_toolkit.key_binding.KeyBindings method), 177
add_binding() (prompt_toolkit.key_binding.KeyBindings method), 177
AdjustBrightnessStyleTransformation (class in prompt_toolkit.styles), 119
AfterInput (class in prompt_toolkit.layout.processors), 163
Always (class in prompt_toolkit.filters), 175
ANSI (class in prompt_toolkit.formatted_text), 96
app (prompt_toolkit.key_binding.key_processor.KeyPressEvent property), 180
append_string() (prompt_toolkit.history.DummyHistory method), 117
append_string() (prompt_toolkit.history.History method), 118
append_string() (prompt_toolkit.history.ThreadedHistory method), 119
append_style_to_content() (prompt_toolkit.layout.screen.Screen method), 168
append_to_arg_count() (prompt_toolkit.key_binding.key_processor.KeyPressEvent method), 180
append_to_history() (prompt_toolkit.buffer.Buffer method), 99
AppendAutoSuggestion (class in prompt_toolkit.layout.processors), 163
Application (class in prompt_toolkit.application), 88
apply_completion() (prompt_toolkit.buffer.Buffer method), 99
apply_search() (prompt_toolkit.buffer.Buffer method), 99
apply_transformation() (prompt_toolkit.layout.processors.BeforeInput method), 164
apply_transformation() (prompt_toolkit.layout.processors.ConditionalProcessor method), 164
apply_transformation() (prompt_toolkit.layout.processors.DisplayMultipleCursors method), 164
apply_transformation() (prompt_toolkit.layout.processors.DummyProcessor method), 164
apply_transformation() (prompt_toolkit.layout.processors.DynamicProcessor method), 165
apply_transformation() (prompt_toolkit.layout.processors.HighlightMatchingBracketProcessor method), 165
apply_transformation() (prompt_toolkit.layout.processors.HighlightSearchProcessor method), 165
apply_transformation() (prompt_toolkit.layout.processors.HighlightSelectionProcessor method), 165
apply_transformation() (prompt_toolkit.layout.processors.PasswordProcessor method), 165
apply_transformation() (prompt_toolkit.layout.processors.Processor method), 166
apply_transformation() (prompt_toolkit.layout.processors.ReverseSearchProcessor method), 166
apply_transformation() (prompt_toolkit.layout.processors.ShowLeadingWhiteSpaceProcessor method), 166
apply_transformation() (prompt_toolkit.layout.processors.ShowTrailingWhiteSpaceProcessor method), 166
apply_transformation() (prompt_toolkit.layout.processors.TabsProcessor method), 167
completions (prompt_toolkit.buffer.CompletionState attribute), 103
CompletionsMenu (class in prompt_toolkit.layout), 163
CompletionState (class in prompt_toolkit.buffer), 103
Condition (class in prompt_toolkit.filters), 175, 176
ConditionalAutoSuggest (class in prompt_toolkit.auto_suggest), 138
ConditionalCompleter (class in prompt_toolkit.completion), 107
ConditionalContainer (class in prompt_toolkit.layout), 152
ConditionalKeyBindings (class in prompt_toolkit.key_binding), 176
ConditionalMargin (class in prompt_toolkit.layout), 161
ConditionalProcessor (class in prompt_toolkit.layout.processors), 164
ConditionalStyleTransformation (class in prompt_toolkit.styles), 120
ConditionalValidator (class in prompt_toolkit.validation), 136
confirm() (in module prompt_toolkit.shortcuts), 130
Container (class in prompt_toolkit.layout), 145
cooked_mode (class in prompt_toolkit.input.vt100), 183
cooked_mode() (prompt_toolkit.input.DummyInput method), 182
cooked_mode() (prompt_toolkit.input.vn100.Vt100Input method), 183
copy_selection() (prompt_toolkit.buffer.Buffer method), 100
cpr_not_supported_callback() (prompt_toolkit.application.Application method), 91
create_app_session() (in module prompt_toolkit.application), 95
create_background_task() (prompt_toolkit.application.Application method), 91
create_confirm_session() (in module prompt_toolkit.shortcuts), 130
create_content() (prompt_toolkit.layout.BufferControl method), 156
create_content() (prompt_toolkit.layout.DummyControl method), 157
create_content() (prompt_toolkit.layout.FormattedTextControl method), 158
create_content() (prompt_toolkit.layout.UIControl method), 158
create_default_formatters() (in module prompt_toolkit.shortcuts.progress_bar.formatters), 135
create_input() (in module prompt_toolkit.input), 183
create_margin() (prompt_toolkit.layout.ConditionalMargin method), 161
create_margin() (prompt_toolkit.layout.Margin method), 160
create_margin() (prompt_toolkit.layout.NumberedMargin method), 161
create_margin() (prompt_toolkit.layout.PromptMargin method), 162
create_margin() (prompt_toolkit.layout.ScrollbarMargin method), 161
create_output() (in module prompt_toolkit.output), 188
create_pipe_input() (in module prompt_toolkit.input), 183
current_buffer (prompt_toolkit.application.Application property), 91
current_buffer (prompt_toolkit.key_binding.key_processor.KeyPressEvent property), 180
current_buffer (prompt_toolkit.layout.Layout property), 144
current_char (prompt_toolkit.document.Document property), 113
current_completion (prompt_toolkit.buffer.CompletionState property), 103
current_control (prompt_toolkit.layout.Layout property), 144
current_line (prompt_toolkit.document.Document property), 113
current_line_after_cursor (prompt_toolkit.document.Document property), 113
current_line_before_cursor (prompt_toolkit.document.Document property), 113
current_search_state (prompt_toolkit.application.Application property), 91
current_window (prompt_toolkit.layout.Layout property), 144
cursor_backward() (prompt_toolkit.output.DummyOutput method), 185
cursor_backward() (prompt_toolkit.output.Output method), 187
cursor_backward() (prompt_toolkit.output.vn100.Vt100_Output method), 189
cursor_down() (prompt_toolkit.buffer.Buffer method), 100
cursor_down() (prompt_toolkit.buffer.DummyOutput method), 185
cursor_down() (prompt_toolkit.output.DummyOutput method), 187
cursor_down() (prompt_toolkit.output.vn100.Vt100_Output method), 189
cursor_forward() (prompt_toolkit.output.DummyOutput method), 185
Index 199
method), 185

cursor_forward() (prompt_toolkit.output.Output method), 187

cursor_forward() (prompt_toolkit.output.vt100.Vt100_Output method), 189

cursor_goto() (prompt_toolkit.output.DummyOutput method), 185
cursor_goto() (prompt_toolkit.output.Vt100_Output method), 187
cursor_goto() (prompt_toolkit.output.vt100.Vt100_Output method), 189
cursor_position (prompt_toolkit.document.Document method), 113
cursor_position_col (prompt_toolkit.document.Document property), 113
cursor_position_row (prompt_toolkit.document.Document property), 114
cursor_positions (prompt_toolkit.layout.screen.Screen attribute), 168
cursor_up() (prompt_toolkit.buffer.Buffer method), 100
cursor_up() (prompt_toolkit.output.DummyOutput method), 185
cursor_up() (prompt_toolkit.output.Output method), 187
cursor_up() (prompt_toolkit.output.vt100.Vt100_Output method), 189
cut_selection() (prompt_toolkit.buffer.Buffer method), 100
cut_selection() (prompt_toolkit.document.Document method), 114

detach() (prompt_toolkit.input.DummyInput method), 182
detach() (prompt_toolkit.input.vt100.Vt100Input method), 183

Dimension (class in prompt_toolkit.layout), 160
disable_autowrap() (prompt_toolkit.output.DummyOutput method), 185
disable_autowrap() (prompt_toolkit.output.Output method), 187
disable_autowrap() (prompt_toolkit.output.vt100.Vt100_Output method), 189
disable_bracketed_paste() (prompt_toolkit.output.DummyOutput method), 185
disable_bracketed_paste() (prompt_toolkit.output.Output method), 187
disable_bracketed_paste() (prompt_toolkit.output.vt100.Vt100_Output method), 190
disable_mouse_support() (prompt_toolkit.output.DummyOutput method), 185
disable_mouse_support() (prompt_toolkit.output.Output method), 187
disable_mouse_support() (prompt_toolkit.output.vt100.Vt100_Output method), 190
display_meta (prompt_toolkit.completion.Completion property), 107
display_meta_text (prompt_toolkit.completion.Completion property), 107
display_text (prompt_toolkit.completion.Completion property), 107
DisplayMultipleCursors (class in prompt_toolkit.layout.processors), 164
Document (class in prompt_toolkit.document), 113
document (prompt_toolkit.buffer.Buffer property), 100
document (prompt_toolkit.widgets.TextArea property), 174
document_for_search() (prompt_toolkit.buffer.Buffer method), 100
draw_all_floats() (prompt_toolkit.layout.screen.Screen method), 168
draw_with_z_index() (prompt_toolkit.layout.screen.Screen method), 168
DummyApplication (class in prompt_toolkit.application), 93
DummyAutoSuggest (class in prompt_toolkit.auto_suggest), 138
DummyClipboard (class in prompt_toolkit.clipboard), 105
DummyCompleter (class in prompt_toolkit.completion), 108
DummyControl (class in prompt_toolkit.layout), 157
DummyHistory (class in prompt_toolkit.history), 117

DeduplicateCompleter (class in prompt_toolkit.completion), 108
default() (prompt_toolkit.output.ColorDepth class method), 184
DEFAULT_BUFFER (in module prompt_toolkit.enums), 117
delete() (prompt_toolkit.buffer.Buffer method), 100
delete_before_cursor() (prompt_toolkit.buffer.Buffer method), 100
DEPTH_1_BIT (prompt_toolkit.output.ColorDepth attribute), 184
DEPTH_24_BIT (prompt_toolkit.output.ColorDepth attribute), 184
DEPTH_4_BIT (prompt_toolkit.output.ColorDepth attribute), 184
DEPTH_8_BIT (prompt_toolkit.output.ColorDepth attribute), 184
detach() (prompt_toolkit.input.DummyInput method), 182
detach() (prompt_toolkit.input.Input method), 182

200 Index
DummyInput (class in prompt_toolkit.input), 182
DummyOutput (class in prompt_toolkit.output), 185
DummyProcessor (class in prompt_toolkit.layout.processors), 164
DummyStyle (class in prompt_toolkit.styles), 121
DummyStyleTransformation (class in prompt_toolkit.styles), 121
DummyValidator (class in prompt_toolkit.validation), 136
DynamicAutoSuggest (class in prompt_toolkit.auto_suggest), 138
DynamicClipboard (class in prompt_toolkit.clipboard), 105
DynamicCompleter (class in prompt_toolkit.completion), 108
DynamicContainer (class in prompt_toolkit.layout), 153
DynamicKeyBindings (class in prompt_toolkit.key_binding), 176
DynamicLexer (class in prompt_toolkit.lexers), 141
DynamicProcessor (class in prompt_toolkit.layout.processors), 164
DynamicStyle (class in prompt_toolkit.styles), 121
DynamicStyleTransformation (class in prompt_toolkit.styles), 122
DynamicValidator (class in prompt_toolkit.validation), 136

EditingMode (class in prompt_toolkit.enums), 117
EditReadOnlyBuffer, 103
empty_line_count_at_the_end() (prompt_toolkit.document.Document method), 114
empty_queue() (prompt_toolkit.key_binding.key_processor.KeyProcessor method), 181
enable_autowrap() (prompt_toolkit.output.DummyOutput method), 185
enable_autowrap() (prompt_toolkit.output.Output method), 187
enable_bracketed_paste() (prompt_toolkit.output.DummyOutput method), 185
enable_bracketed_paste() (prompt_toolkit.output.Output method), 187
enable_bracketed_paste() (prompt_toolkit.output.vt100.Vt100_Output method), 190
enable_mouse_support() (prompt_toolkit.output.DummyOutput method), 190
enable_mouse_support() (prompt_toolkit.output.DummyOutput method), 185
enable_mouse_support() (prompt_toolkit.output.Output method), 187
enable_mouse_support() (prompt_toolkit.output.vt100.Vt100_Output method), 190
encoding() (prompt_toolkit.output.DummyOutput method), 185
encoding() (prompt_toolkit.output.DummyOutput method), 185
encoding() (prompt_toolkit.output.Output method), 187
encoding() (prompt_toolkit.output.vt100.Vt100_Output method), 190
end_of_paragraph() (prompt_toolkit.document.Document method), 114
enter_alternate_screen() (prompt_toolkit.output.DummyOutput method), 185
enter_alternate_screen() (prompt_toolkit.output.DummyOutput method), 187
enter_alternate_screen() (prompt_toolkit.output.Output method), 187
enter_alternate_screen() (prompt_toolkit.output.vt100.Vt100_Output method), 190
erase() (prompt_toolkit.renderer.Renderer.Renderer method), 140
erase() (prompt_toolkit.output.DummyOutput method), 185
erase() (prompt_toolkit.output.DummyOutput method), 185
erase() (prompt_toolkit.output.Output method), 187
erase() (prompt_toolkit.output.vt100.Vt100_Output method), 190
erase_end_of_line() (prompt_toolkit.output.DummyOutput method), 185
erase_end_of_line() (prompt_toolkit.output.DummyOutput method), 187
erase_end_of_line() (prompt_toolkit.output.Output method), 187
erase_end_of_line() (prompt_toolkit.output.vt100.Vt100_Output method), 190
erase_screen() (prompt_toolkit.output.DummyOutput method), 185
erase_screen() (prompt_toolkit.output.DummyOutput method), 185
erase_screen() (prompt_toolkit.output.Output method), 187
erase_screen() (prompt_toolkit.output.vt100.Vt100_Output method), 190
exact() (prompt_toolkit.layout.Dimension class method), 160
ExecutableCompleter (class in prompt_toolkit.completion), 109
exit() (prompt_toolkit.application.Application method), 91
explode_text_fragments() (in module prompt_toolkit.layout.utils), 168

F
feed() (prompt_toolkit.input.vt100_parser.Vt100Parser method), 184
feed() (prompt_toolkit.input.vt100_parser.Vt100Parser method), 185
feed() (prompt_toolkit.key_binding.key_processor.KeyProcessor method), 181

Index 201
feed_and_flush() (prompt_toolkit.input.vt100_parser.Vt100Parser(prompt_toolkit.document.Document method), 115
feed_multiple() (prompt_toolkit.key_binding.key_processor.KeyProcessor method), 184
find_start_of_previous_word() (prompt_toolkit.document.Document method), 115
FileHistory (class in prompt_toolkit.history), 118
fileno() (prompt_toolkit.input.DummyInput method), 182
fileno() (prompt_toolkit.input.Input method), 182
fileno() (prompt_toolkit.input.vt100.Vt100Input method), 183
fileno() (prompt_toolkit.output.DummyOutput method), 186
fileno() (prompt_toolkit.output.Output method), 187
fileno() (prompt_toolkit.output.vt100.Vt100_Output method), 190
fill_area() (prompt_toolkit.layout.screen.Screen method), 168
Filter (class in prompt_toolkit.filters), 175
find() (prompt_toolkit.document.Document method), 114
find_all() (prompt_toolkit.document.Document method), 114
find_all_windows() (prompt_toolkit.layout.Layout method), 144
find_backwards() (prompt_toolkit.document.Document method), 114
find_boundaries_of_current_word() (prompt_toolkit.document.Document method), 114
find_enclosing_bracket_left() (prompt_toolkit.document.Document method), 114
find_enclosing_bracket_right() (prompt_toolkit.document.Document method), 114
find_matching_bracket_position() (prompt_toolkit.document.Document method), 114
find_next_matching_line() (prompt_toolkit.document.Document method), 114
find_next_word_beginning() (prompt_toolkit.document.Document method), 114
find_next_word_ending() (prompt_toolkit.document.Document method), 114
find_previous_matching_line() (prompt_toolkit.document.Document method), 115
find_previous_word_beginning() (prompt_toolkit.document.Document method), 115
find_previous_word_ending() (prompt_toolkit.document.Document method), 115
Float (class in prompt_toolkit.layout), 149
FloatContainer (class in prompt_toolkit.layout), 148
flush() (prompt_toolkit.input.DummyInput method), 182
flush() (prompt_toolkit.input.vt100_parser.Vt100Parser method), 184
flush() (prompt_toolkit.output.DummyOutput method), 186
flush() (prompt_toolkit.output.Output method), 187
flush() (prompt_toolkit.output.vt100.Vt100_Output method), 190
flush() (prompt_toolkit.patch_stdout.StdoutProxy method), 192
flush_keys() (prompt_toolkit.input.DummyInput method), 182
flush_keys() (prompt_toolkit.input.vt100.Vt100Input method), 183
focus() (prompt_toolkit.layout.Layout method), 144
focus_last() (prompt_toolkit.layout.Layout method), 144
focus_next() (prompt_toolkit.layout.Layout method), 144
focus_previous() (prompt_toolkit.layout.Layout method), 144
format() (prompt_toolkit.formatted_text.ANSI method), 96
format() (prompt_toolkit.formatted_text.HTML method), 96
FormattedText (class in prompt_toolkit.formatted_text), 96
FormattedTextControl (class in prompt_toolkit.layout), 157
FormattedTextToolBar (class in prompt_toolkit.widgets), 170
Formatter (class in prompt_toolkit.shortcuts.progress_bar.formatters), 135
fragment_list_len() (in module prompt_toolkit.formatted_text), 97
fragment_list_to_text() (in module prompt_toolkit.formatted_text), 97
fragment_list_width() (in module prompt_toolkit.formatted_text), 97
Frame (class in prompt_toolkit.widgets), 170
from_callable() (prompt_toolkit.validation.Validator class method), 137
from_dict() (prompt_toolkit.styles.Style class method), 122
from_env() (prompt_toolkit.output.ColorDepth class method), 184
from_filename() (prompt_toolkit.lexers.PygmentsLexer class method), 142
get_data() (prompt_toolkit.clipboard.DummyClipboard method), 104
get_data() (prompt_toolkit.clipboard.DynamicClipboard method), 105
get_data() (prompt_toolkit.clipboard.InMemoryClipboard method), 105
get_data() (prompt_toolkit.clipboard.pyperclip.PyperclipClipboard method), 106
get_default_color_depth() (prompt_toolkit.output.DummyOutput method), 186
get_default_color_depth() (prompt_toolkit.output.Output method), 187
get_default_color_depth() (prompt_toolkit.output.vt100.Vt100_Output method), 190
get_end_of_line_position() (prompt_toolkit.document.Document method), 115
get_event_loop() (in module prompt_toolkit.eventloop), 181
get_focusable_windows() (prompt_toolkit.layout.Layout method), 144
get_height_for_line() (prompt_toolkit.layout.UIContent method), 159
get_invalidate_events() (prompt_toolkit.layout.BufferControl method), 156
get_invalidate_events() (prompt_toolkit.layout.UIControl method), 158
get_key_bindings() (prompt_toolkit.layout.BufferControl method), 156
get_key_bindings() (prompt_toolkit.layout.Container method), 145
get_key_bindings() (prompt_toolkit.layout.DynamicContainer method), 153
get_key_bindings() (prompt_toolkit.layout.FloatContainer method), 149
get_key_bindings() (prompt_toolkit.layout.FormattedTextControl method), 158
get_key_bindings() (prompt_toolkit.layout.ScrollablePane method), 154
get_key_bindings() (prompt_toolkit.layout.UIControl method), 159
get_key_bindings() (prompt_toolkit.layout.Window method), 152
get_menu_position() (prompt_toolkit.layout.screen.Screen method), 204
prompt_toolkit.eventloop), 181
get_used_style_strings()
(prompt_toolkit.application.Application method), 91
get_visible_focusable_windows()
(prompt_toolkit.layout.Layout method), 145
get_width()
(prompt_toolkit.layout.ConditionalMargin method), 162
get_width()
(prompt_toolkit.layout.Margin method), 160
get_width()
(prompt_toolkit.layout.NumberedMargin method), 161
get_width()
(prompt_toolkit.layout.PromptMargin method), 162
get_width()
(prompt_toolkit.layout.ScrollbarMargin method), 161
get_word_before_cursor()
(prompt_toolkit.document.Document method), 115
get_word_under_cursor()
(prompt_toolkit.document.Document method), 115
go_to_completion()
(prompt_toolkit.buffer.Buffer method), 100
go_to_history()
(prompt_toolkit.buffer.Buffer method), 100
go_to_index()
(prompt_toolkit.buffer.CompletionState method), 103

has_focus() (in module prompt_toolkit.filters), 175
has_focus() (in module prompt_toolkit.filters.app), 176
has_focus() (prompt_toolkit.layout.Layout method), 145
has_match_at_current_position()
(prompt_toolkit.document.Document method), 116
HasFocus() (in module prompt_toolkit.filters), 175
height_is_known
(prompt_toolkit.renderer.Renderer property), 140
hidden
(prompt_toolkit.styles.Attrs property), 120
hide_cursor()
(prompt_toolkit.output.DummyOutput method), 186
hide_cursor()
(prompt_toolkit.output.Output method), 188
hide_cursor()
(prompt_toolkit.output.vt100.Vt100_Output method), 190
HighlightIncrementalSearchProcessor
(class in prompt_toolkit.layout.processors), 165
HighlightMatchingBracketProcessor
(class in prompt_toolkit.layout.processors), 165
HighlightSearchProcessor
(class in prompt_toolkit.layout.processors), 165

History (class in prompt_toolkit.history), 118
history_backward()
(prompt_toolkit.buffer.Buffer method), 100
history_forward()
(prompt_toolkit.buffer.Buffer method), 101
HorizontalAlign (class in prompt_toolkit.layout), 155
HorizontalLine (class in prompt_toolkit.widgets), 170
HSplit (class in prompt_toolkit.layout), 146
HTML (class in prompt_toolkit.formatted_text), 96

in_editing_mode() (in module prompt_toolkit.filters), 175
in_editing_mode() (in module prompt_toolkit.filters.app), 176
in_terminal() (in module prompt_toolkit.application), 95
indent() (module prompt_toolkit.buffer), 103
InEditingMode() (module prompt_toolkit.filters), 175
InMemoryClipboard
(class in prompt_toolkit.clipboard), 105
InMemoryHistory (class in prompt_toolkit.history), 118
Input (module prompt_toolkit.input), 182
input_dialog() (module prompt_toolkit.shortcuts), 130
input_mode (module prompt_toolkit.key_binding.vi_state.ViState property), 179
InputMode (class in prompt_toolkit.key_binding.vi_state), 179
insert_after()
(prompt_toolkit.document.Document method), 116
insert_before()
(prompt_toolkit.document.Document method), 116
insert_line_above()
(prompt_toolkit.buffer.Buffer method), 101
insert_line_below()
(prompt_toolkit.buffer.Buffer method), 101
insert_text()
(prompt_toolkit.buffer.Buffer method), 101
invalidate()
(prompt_toolkit.application.Application method), 92
invalidated
(prompt_toolkit.application.Application property), 92
invalidation_hash()
(prompt_toolkit.lexers.DynamicLexer method), 141
invalidation_hash()
(prompt_toolkit.lexers.Lexer method), 141
invalidation_hash()
(prompt_toolkit.styles.AdjustBrightnessStyleTransformation method), 119

Index 205
invalidation_hash() (prompt_toolkit.styles.BaseStyle method), 120
invalidation_hash() (prompt_toolkit.styles.ConditionalStyleTransformation method), 121
invalidation_hash() (prompt_toolkit.styles.DummyStyle method), 121
invalidation_hash() (prompt_toolkit.styles.DummyStyleTransformation method), 121
invalidation_hash() (prompt_toolkit.styles.DynamicStyle method), 122
invalidation_hash() (prompt_toolkit.styles.DynamicStyleTransformation method), 122
invalidation_hash() (prompt_toolkit.styles.Style method), 123
invalidation_hash() (prompt_toolkit.styles.StyleTransformation method), 123
InvalidLayoutError (class in prompt_toolkit.layout), 145
is_container (class in prompt_toolkit.layout), 155
is_cursor_at_the_end (prompt_toolkit.document.Document property), 116
is_cursor_at_the_end_of_line (prompt_toolkit.document.Document property), 116
is_focusable() (prompt_toolkit.layout.BufferControl method), 156
is_focusable() (prompt_toolkit.layout.DummyControl method), 157
is_focusable() (prompt_toolkit.layout.FormattedTextControl method), 158
is_focusable() (prompt_toolkit.layout.UIControl method), 159
is_formatted_text() (in module prompt_toolkit.formatted_text), 97
is_modal() (prompt_toolkit.layout.Container method), 145
is_modal() (prompt_toolkit.layout.DynamicContainer method), 153
is_modal() (prompt_toolkit.layout.FloatContainer method), 149
is_modal() (prompt_toolkit.layout.ScrollablePane method), 154
is_repeat (prompt_toolkit.key_binding.key_processor.KeyPressEvent attribute), 180
is_returnable (prompt_toolkit.buffer.Buffer property), 101
is_running (prompt_toolkit.application.Application property), 92
is_searching (prompt_toolkit.layout.Layout property), 145
is_true() (in module prompt_toolkit.filters), 175
is_true() (in module prompt_toolkit.filters.utils), 176
is_zero() (prompt_toolkit.layout.Dimension method), 160
italic (prompt_toolkit.styles.Attrs property), 120
IterationsPerSecond (class in prompt_toolkit.shortcuts.progress_bar.formatters), 135
join_next_line() (prompt_toolkit.buffer.Buffer method), 101
join_selected_lines() (prompt_toolkit.buffer.Buffer method), 101
key_processor (prompt_toolkit.application.Application attribute), 92
KeyBindings (class in prompt_toolkit.key_binding), 177
KeyBindingsBase (class in prompt_toolkit.key_binding), 178
KeyPress (class in prompt_toolkit.key_binding.key_processor), 180
KeyPressEvent (class in prompt_toolkit.key_binding.key_processor), 180
KeyProcessor (class in prompt_toolkit.key_binding.key_processor), 180
Keys (class in prompt_toolkit.keys), 119
Label (class in prompt_toolkit.shortcuts.progress_bar.formatters), 135
Label (class in prompt_toolkit.widgets), 170
last_character_find (prompt_toolkit.key_binding.vi_state.ViState attribute), 179
last_non_blank_of_current_line_position() (prompt_toolkit.document.Document property), 116
last_rendered_screen (prompt_toolkit.renderer.Renderer property), 140
Layout (class in prompt_toolkit.layout), 144
leading_whitespace_in_current_line (prompt_toolkit.document.Document property), 116
lex_document() (prompt_toolkit.lexers.DynamicLexer method), 141
lex_document() (prompt_toolkit.lexers.Lexer method), 141
lex_document() (prompt_toolkit.lexers.Pygments Lexer method), 142
lex_document() (prompt_toolkit.lexers.SimpleLexer method), 142
Lexer (class in prompt_toolkit.lexers), 141
line_count (prompt_toolkit.document.Document property), 116
lines (prompt_toolkit.document.Document property), 116
LINES (prompt_toolkit.selection.SelectionType attribute), 104
lines_from_current (prompt_toolkit.document.Document property), 116
load() (prompt_toolkit.history.History method), 118
load() (prompt_toolkit.history.ThreadedHistory method), 119
load_history_if_not_yet_loaded() (prompt_toolkit.buffer.Buffer method), 101
load_history_strings() (prompt_toolkit.history.DummyHistory method), 117
load_history_strings() (prompt_toolkit.history.FileHistory method), 118
load_history_strings() (prompt_toolkit.history.InMemoryHistory method), 118
load_history_strings() (prompt_toolkit.history.ThreadedHistory method), 119
load_key_bindings() (in module prompt_toolkit.key_binding.defaults), 179

M

Margin (class in prompt_toolkit.layout), 160
menu_positions (prompt_toolkit.layout.screen.Screen attribute), 168
MenuContainer (class in prompt_toolkit.layout.screen.Attribute), 171
merge_completers() (in module prompt_toolkit.completion), 113
merge_formatted_text() (in module prompt_toolkit.formatted_text), 97
merge_key_bindings() (in module prompt_toolkit.key_binding.defaults), 179
merge_processors() (in module prompt_toolkit.layout.processors), 167
merge_style_transformations() (in module prompt_toolkit.styles), 123
merge_styles() (in module prompt_toolkit.styles), 123
message_dialog() (in module prompt_toolkit.shortcuts), 130
mouse_handler() (prompt_toolkit.layout.BufferControl method), 156
mouse_handler() (prompt_toolkit.layout.FormattedTextControl method), 158
mouse_handler() (prompt_toolkit.layout.UIManager method), 159
move_cursor_down() (prompt_toolkit.layout.BufferControl method), 156
move_cursor_down() (prompt_toolkit.layout.UIManager method), 159
validate() (prompt_toolkit.buffer.Buffer method), 103
validate() (prompt_toolkit.validation.ConditionalValidator method), 136
validate() (prompt_toolkit.validation.DummyValidator method), 136
validate() (prompt_toolkit.validation.DynamicValidator method), 136
validate() (prompt_toolkit.validation.ThreadedValidator method), 136
validate() (prompt_toolkit.validation.Validator method), 137
validate_and_handle() (prompt_toolkit.buffer.Buffer method), 103
validate_async() (prompt_toolkit.validation.DynamicValidator method), 136
validate_async() (prompt_toolkit.validation.ThreadedValidator method), 136
validate_async() (prompt_toolkit.validation.Validator method), 137
ValidationError, 136
Validator (class in prompt_toolkit.validation), 136
VerticalAlign (class in prompt_toolkit.layout), 155
VerticalLine (class in prompt_toolkit.widgets), 174
vi_state (prompt_toolkit.application.Application attribute), 93
ViState (class in prompt_toolkit.key_binding.vi_state), 179
VSplit (class in prompt_toolkit.layout), 147
Vt100_Output (class in prompt_toolkit.output.vt100), 189
Vt100Input (class in prompt_toolkit.input.vt100), 183
Vt100Parser (class in prompt_toolkit.input.vt100_parser), 183

write() (prompt_toolkit.output.DummyOutput method), 186
write() (prompt_toolkit.output.Output method), 188
write() (prompt_toolkit.output.vt100.Vt100_Output method), 191
write_raw() (prompt_toolkit.output.DummyOutput method), 186
write_raw() (prompt_toolkit.output.Output method), 188
write_raw() (prompt_toolkit.output.vt100.Vt100_Output method), 191
write_to_screen() (prompt_toolkit.layout.ConditionalContainer method), 152
write_to_screen() (prompt_toolkit.layout.Container method), 146
write_to_screen() (prompt_toolkit.layout.DynamicContainer method), 153
write_to_screen() (prompt_toolkit.layout.FloatContainer method), 149
write_to_screen() (prompt_toolkit.layout.HSplit method), 147
write_to_screen() (prompt_toolkit.layout.ScrollablePane method), 155
write_to_screen() (prompt_toolkit.layout.VSplit method), 148
write_to_screen() (prompt_toolkit.layout.Window method), 152

yank_last_arg() (prompt_toolkit.buffer.Buffer method), 103
yank_nth_arg() (prompt_toolkit.buffer.Buffer method), 103
yes_no_dialog() (in module prompt_toolkit.shortcuts), 134

zero() (prompt_toolkit.layout.Dimension class method), 160
zero_width_escapes (prompt_toolkit.layout.screen.Screen attribute), 169

Index