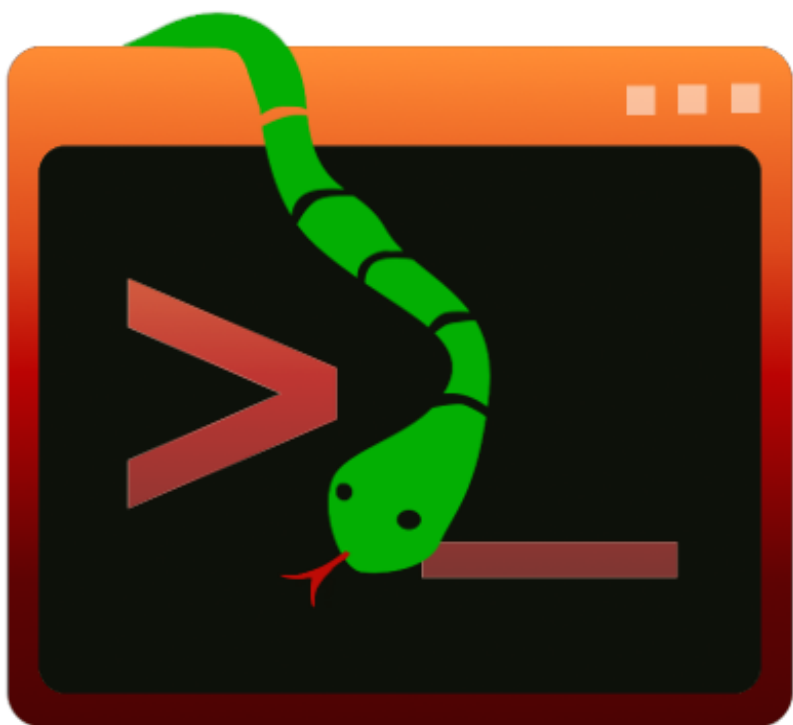


# PYTHON PROMPT TOOLKIT 3.0



# Chapter 1

## Python Prompt toolkit 3.0

**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.)
- Multiline input editing.
- Advanced code completion.
- Selecting text for copy/paste. (Both **Emacs** and **Vi** style.)
- Mouse support for cursor positioning and scrolling.
- Auto suggestions.
- 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.

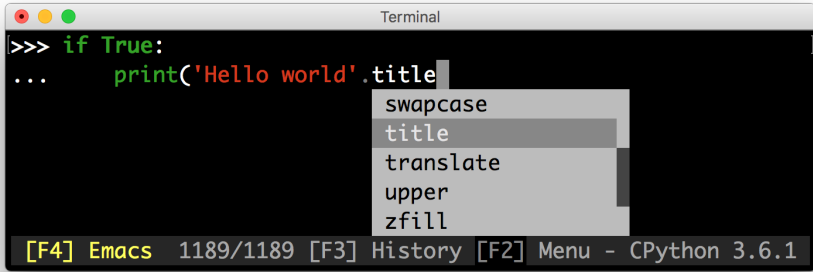


Figure 1. Ptpython code completion.

- 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.

# Chapter 2

## Getting started

### Installation

Install **Prompt toolkit** with packet manager or **Pip**:

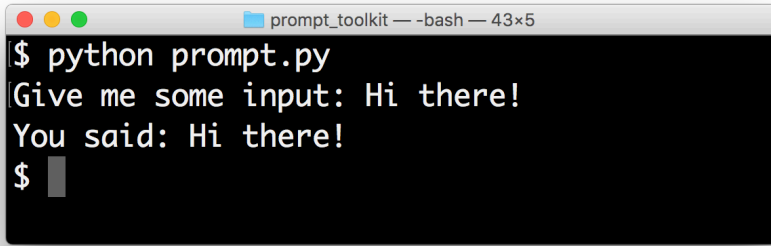
```
pip install prompt_toolkit
```

### 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.



```
prompt_toolkit -- -bash -- 43x5
$ python prompt.py
Give me some input: Hi there!
You said: Hi there!
$
```

**Figure 2.** A simple prompt.

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 application.

## A simple prompt

The following snippet is the most simple example, it uses the `prompt_toolkit.shortcuts.prompt` function to ask the user for input and returns the text. Just like the `input` function.

```
from prompt_toolkit import prompt

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

## Chapter 3

# Printing (and using) formatted text

**Prompt toolkit** ships with a `prompt_toolkit.shortcuts.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 information 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 `prompt_toolkit.shortcuts.print_formatted_text` takes any kind of *formatted text* as input, prompts and toolbars also accept *formatted text*.

## Printing plain text

The `print` function can be imported as follows:

```
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.

```
> from prompt_toolkit import print_formatted_text, HTML
>
> s = (
|     '<b>Bold</b>, <i>italic</i>, <u>underlined</u>, '
|     '<ansired>Red</ansired>, <ansigreen>green</ansigreen>, '
|     '<skyblue>sky blue</skyblue>, \n'
|     '<seagreen>sea green</seagreen>, '
|     '<violet>violet</violet>, '
|     '<aaa fg="ansiwhite" bg="ansigreen">white on green</aaa>')
>
> print_formatted_text (HTML (s))
Bold, italic, underlined, Red, green, sky blue,
sea green, violet, white on green
```

Figure 3. HTML formatted text.

```
from prompt_toolkit import print_formatted_text as print

print ('Hello world')
```

## Formatted text

There are several ways to display colors:

- By creating a `prompt_toolkit.formatted_text.HTML` object.
- By creating a `prompt_toolkit.formatted_text.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 `prompt_toolkit.formatted_text.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

The class `prompt_toolkit.formatted_text.HTML` can be used to indicate that a string contains HTML-like formatting. It recognizes the basic tags for bold, italic

and underline: `<b>`, `<i>` and `<u>`.

```
from prompt_toolkit import print_formatted_text, HTML

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

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

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

# Named colors
# 256 color palette, or true color, depending on the output

print_formatted_text (HTML (
    '<skyblue>This is sky blue</skyblue>'))
print_formatted_text (HTML (
    '<seagreen>This is sea green</seagreen>'))
print_formatted_text (HTML (
    '<violet>This is violet</violet>'))
```

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

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

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

```
from 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 `prompt_toolkit.formatted_text.ANSI` class takes care of that.

```

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.

## The (style, text) tuples

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

```

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

```
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 `prompt_toolkit.formatted_text.PygmentsTokens` object.

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

```
text = [
```

```

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

```

```
print_formatted_text (PygmentsTokens (text))
```

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

```

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

from prompt_toolkit.formatted_text import PygmentsTokens
from prompt_toolkit import print_formatted_text

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

```

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

<b>Pygments</b> token	Prompt toolkit classname (with "class:" prefix)
Token.Keyword	"pygments.keyword"
Token.Punctuation	"pygments.punctuation"
Token.Literal.String. Double	"pygments.literal.string.double"
Token.Text	"pygments.text"
Token	"pygments"

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

```
from prompt_toolkit.styles import Style

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

## The `to_formatted_text` function

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

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

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

print_formatted_text (text)
```

## Chapter 4

# Asking for input (prompts)

This chapter 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 chapter “*Building full screen applications*”.

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

## Hello world

The following snippet is the most simple example, it uses the `prompt_toolkit.shortcuts.prompt` function to ask the user for input and returns the text. Just like the `input` function.

```
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_toolkit.shortcuts.prompt` has a lot of configuration options. In the following sections, we will discover all these parameters.

## The PromptSession object

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

```
from prompt_toolkit import PromptSession
```

```
# Create prompt object:  
session = PromptSession ()
```

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

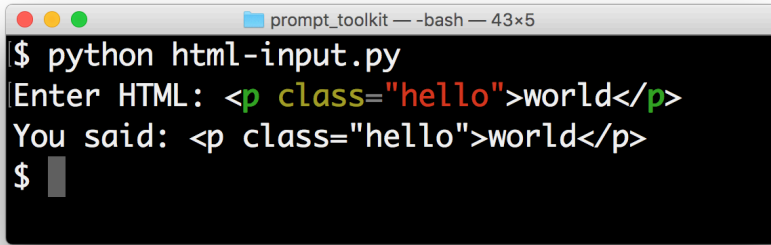
This has mainly two advantages:

- The input history will be kept between consecutive `prompt_toolkit.shortcuts.PromptSession.prompt` calls.
- The `prompt_toolkit.shortcuts.PromptSession` instance and its `prompt_toolkit.shortcuts.PromptSession.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 `prompt_toolkit.shortcuts.PromptSession` instance as well, and they can be overridden by passing values to the `prompt_toolkit.shortcuts.PromptSession.prompt` method.

## 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 `prompt_toolkit.lexers.PygmentsLexer`. It is also possible to create a custom lexer by implementing the `prompt_toolkit.lexers.Lexer` abstract base class.

```
from pygments.lexers.html import HtmlLexer  
from prompt_toolkit.shortcuts import prompt
```



```
$ python html-input.py
Enter HTML: <p class="hello">world</p>
You said: <p class="hello">world</p>
$
```

Figure 4. A simple prompt with syntax highlighting.

```
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 our custom **Pygments** style doesn't specify a color.

## Colors

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

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

our_style = Style.from_dict ({
    'pygments.comment':    '#888888 bold',
    'pygments.keyword':    '#ff88ff bold',
})

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

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

- The `roman`, `sans`, `mono` and `border` options are ignored.
- The style has a few additions: `blink`, `noblink`, `reverse` and `noreverse`.
- Colors can be in the `#ff0000` format, but they can be one of the built-in ANSI color names as well. In that case, they map directly to the 16-color palette of the terminal.

Read more about styling in chapter “*Styling*”.



## Using a Pygments style

All **Pygments** style classes can be used as well, when they are wrapped through `prompt_toolkit.styles.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.

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

style = Style.from_dict ({
```

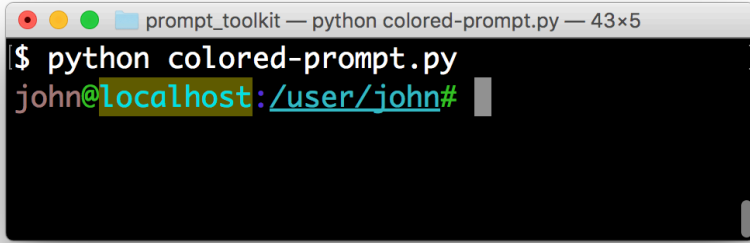


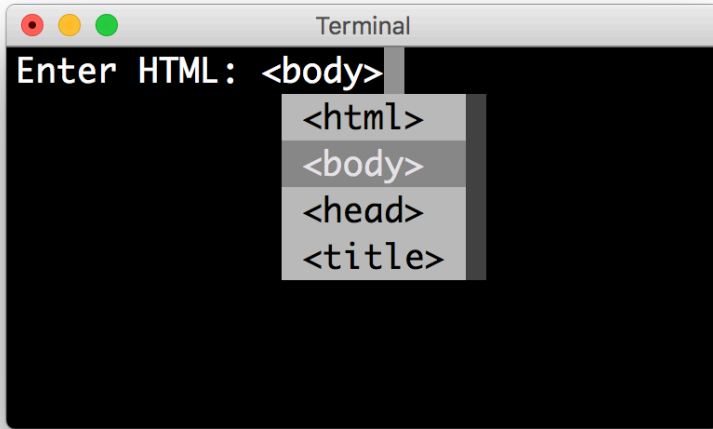
Figure 5. A colored prompt.

```
# 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)
```



**Figure 6.** A prompt with HTML completion.

The message can be any kind of *formatted text*, as discussed in chapter “*Formatted text*”. 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 24-bit true color, this is possible by adding the `color_depth=ColorDepth.TRUE_COLOR` option to the `prompt_toolkit.shortcuts.prompt.prompt` function.

```
from prompt_toolkit.output import ColorDepth
```

```
text = prompt (
    message, style=style,
    color_depth=ColorDepth.TRUE_COLOR)
```

## Autocompletion

Autocompletion can be added by passing a `completer` parameter. This should be an instance of the `prompt_toolkit.completion.Completer` abstract base

class. The `prompt_toolkit.completion.WordCompleter` is an example of a completer that implements that interface.

```
from prompt_toolkit import prompt
from prompt_toolkit.completion import WordCompleter

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

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

**Note:** In **Prompt toolkit 2.0**, the autocompletion 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 `prompt_toolkit.completion.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. A simple `prompt_toolkit.completion.WordCompleter` is not enough in that case. We want to be able to define completions at multiple hierarchical levels. The `prompt_toolkit.completion.NestedCompleter` solves this issue:

```
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,
})

text = prompt ('# ', completer=completer)
print ('You said: %s' % text)

```

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:

```

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 `prompt_toolkit.completion.Completer` class has to implement a generator named `prompt_toolkit.completion.Completer.get_completions` that takes a `prompt_toolkit.document.Document` and yields the current `prompt_toolkit.completion.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.

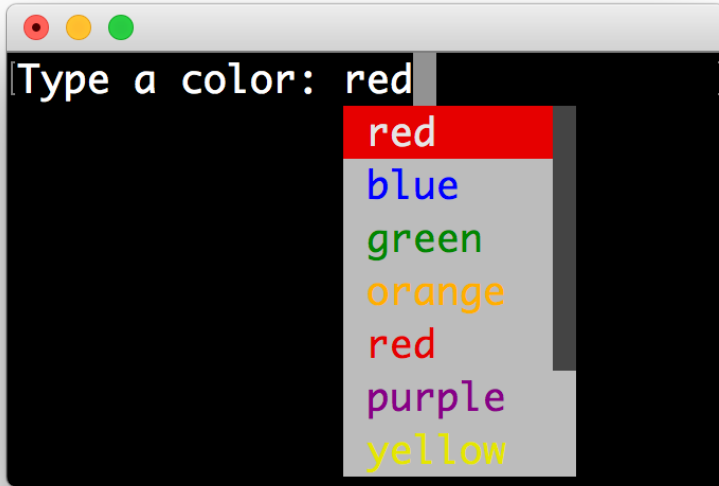


Figure 7. Colorfur completion.

## 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 `prompt_toolkit.completion.Completion` instance.

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

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

```

yield Completion ('completion2', start_position=0,
                  style='underline')

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

```

The next example uses completion styling.

```

from prompt_toolkit.completion import Completer, Completion
from prompt_toolkit.output.color_depth import ColorDepth
from prompt_toolkit.shortcuts import CompleteStyle, prompt

```

```

colors = [
    "red", "blue", "green", "orange", "purple",
    "yellow", "cyan", "magenta", "pink", ]

```

```

class ColorCompleter (Completer):
    def get_completions (self, document, complete_event):
        word = document.get_word_before_cursor ()
        for color in colors:
            if color.startswith (word):
                yield Completion (
                    color,
                    start_position=-len(word),
                    style="fg:" + color,
                    selected_style="fg:white bg:" + color,
                )

```

```

print ("(The completion menu displays colors.)")
prompt ("Type a color: ", completer=ColorCompleter())

```

Finally, it is possible to pass *formatted text* for the display attribute of a `prompt_toolkit.completion.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:

```

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><ansired>1</ansired>'),
            style='bg:ansiyellow')

```

## Fuzzy completion

If one possible completion 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 `prompt_toolkit.completion.FuzzyCompleter` and `prompt_toolkit.completion.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 `prompt_toolkit.completion.WordCompleter` wrapped into a `prompt_toolkit.completion.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:

```

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





**Figure 8.** Number validator.

`prompt_toolkit.completion.ThreadedCompleter`, but also by passing the `complete_in_thread=True` argument.

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

## 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 implement the `prompt_toolkit.validation.Validator` abstract base class. This requires only one method, named `validate` that takes a `prompt_toolkit.document.Document` as input and raises `prompt_toolkit.validation.ValidationError` when the validation fails.

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

class NumberValidator (Validator):
    def validate (self, document):
```

```

text = document.text

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

    # Get index of 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:

```

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 `prompt_toolkit.validation.ThreadedValidator`.

## Validator from a callable

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

```

from prompt_toolkit.validation import Validator
from prompt_toolkit import prompt

```

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

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

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

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

## History

A `prompt_toolkit.history.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 `prompt_toolkit.shortcuts.PromptSession`, which uses a `prompt_toolkit.history.InMemoryHistory` for the entire session by default. The following example has a history out of the box:

```
from prompt_toolkit import PromptSession

session = PromptSession ()

while True:
    session.prompt ()
```

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



Figure 9. The history page with its help. (This is a full screen layout.)

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

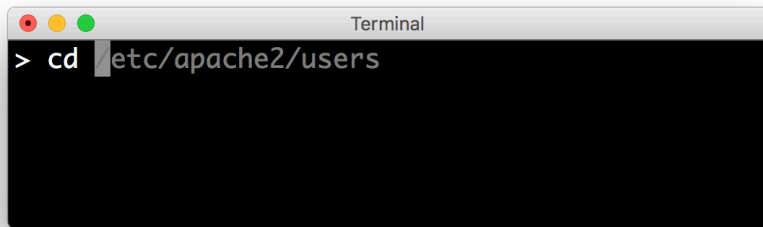
session = PromptSession (
    history=FileHistory('~/.myhistory'))

while True:
    session.prompt ()
```

## 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 cur-



**Figure 10.** Auto suggestion.

rent input. Pressing the right arrow → or **Control-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 `prompt_toolkit.history.History` object between consecutive `prompt_toolkit.shortcuts.prompt` calls. Using a `prompt_toolkit.shortcuts.PromptSession` does this for you.

Example:

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

session = PromptSession ()

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

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

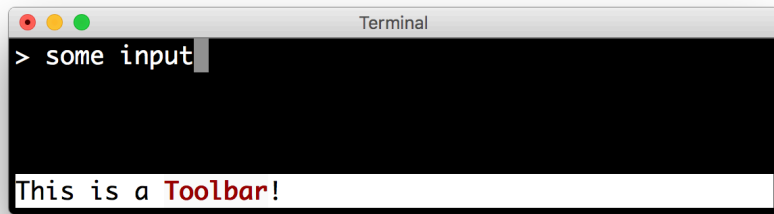


Figure 11. Bottom toolbar

## Adding a bottom toolbar

Adding a bottom toolbar is as easy as passing a `bottom_toolbar` argument to `prompt_toolkit.shortcuts.prompt`. This argument can 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 a `prompt_toolkit.formatted_text.HTML` object. By default, the toolbar has the *reversed* style, which is why we are setting the background instead of the foreground.

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

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

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

Similar, we could use a list of `(style, text)` tuples.

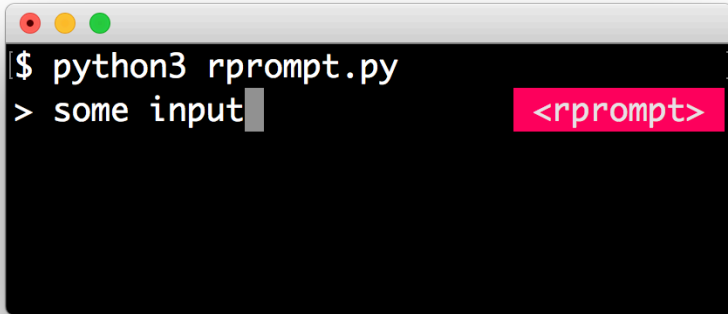


Figure 12. Right prompt.

```
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.

## Adding a right prompt

The `prompt_toolkit.shortcuts.prompt` function has out of the box support for right prompts as well. People familiar to **ZSH** could recognise this as the **R\_PROMPT** option.

So, similar to adding a bottom toolbar, we can pass a `rprompt` argument. This can be either *plain text*, *formatted text* or a *callable* which returns either.

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

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

def get_rprompt ():
    return '<rprompt>'

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

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

## Vi input mode

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

```
from prompt_toolkit import prompt

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



## 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 `prompt_toolkit.key_binding.KeyBindings` instance to the `key_bindings` argument of the `prompt_toolkit.shortcuts.prompt` function or the `prompt_toolkit.shortcuts.PromptSession` class.

An example of a prompt that prints “hello world” when **Control-T** is pressed.

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

bindings = KeyBindings ()

@bindings.add ('c-t')
def _ (event):
    " Say 'hello' when `c-t` is pressed. "
    def print_hello ():
        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 `prompt_toolkit.application.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 `prompt_toolkit.filters.Filter`, usually a `prompt_toolkit.filters.Condition` instance. (Read more about filters in chapter “Filters”.)

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

bindings = KeyBindings ()

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

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

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

## Dynamically switch between Emacs and Vi mode

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

```
from prompt_toolkit import prompt
from prompt_toolkit.application.current import get_app
from prompt_toolkit.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)
        ]

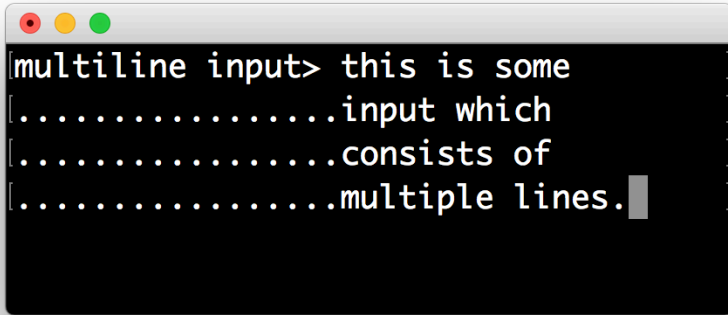
    prompt ('> ', key_bindings=bindings,
            bottom_toolbar=bottom_toolbar)
run ()

```

Read more about key bindings in chapter “*Key bindings*”.

## Using Control-Space for completion

A 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



**Figure 13.** Multiline input.

following key binding.

```
kb = KeyBindings ()

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

## Other prompt options

### Multiline input

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

```
from prompt_toolkit import prompt
```

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

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

It is possible to specify a *continuation prompt*. This works by passing a `prompt_continuation callable` to `prompt_toolkit.shortcuts.prompt`. This function is supposed to return *formatted text*, or a list of (style,text) tuples. The width of the returned text should not exceed the given width. (The width of the prompt margin is defined by the prompt.)

```
from prompt_toolkit import prompt

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

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

## Passing a default value

A default value can be given:

```
from prompt_toolkit import prompt
import getpass

prompt ('What is your name: ', default=f'{getpass.getuser()}')
```

## Mouse support

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

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

```
from prompt_toolkit import prompt
```

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

## Line wrapping

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

```
from prompt_toolkit import prompt

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

## Password input

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

```
from prompt_toolkit import prompt

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

## 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.

```
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)
prompt ('>', cursor=CursorShape.BLINKING_BLOCK)
prompt ('>', cursor=CursorShape.BLINKING_UNDERLINE)
```

```
prompt ('>', cursor=CursorShape.BLINKING_BEAM)
prompt ('>', cursor=ModalCursorShapeConfig())
```

## Prompt in an Asyncio application

For **Asyncio** applications, it's very important to never block the eventloop. However, `prompt_toolkit.shortcuts.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_toolkit.shortcuts.PromptSession.prompt_async` instead of `prompt_toolkit.shortcuts.PromptSession.prompt`. The `async` variation returns a coroutine and is awaitable.

```
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 `prompt_toolkit.patch_stdout.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.

## 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:

```
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.



# Chapter 5

## Dialogs

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

### Message box

Use the `prompt_toolkit.shortcuts.message_dialog` function to display a simple message box. For instance:

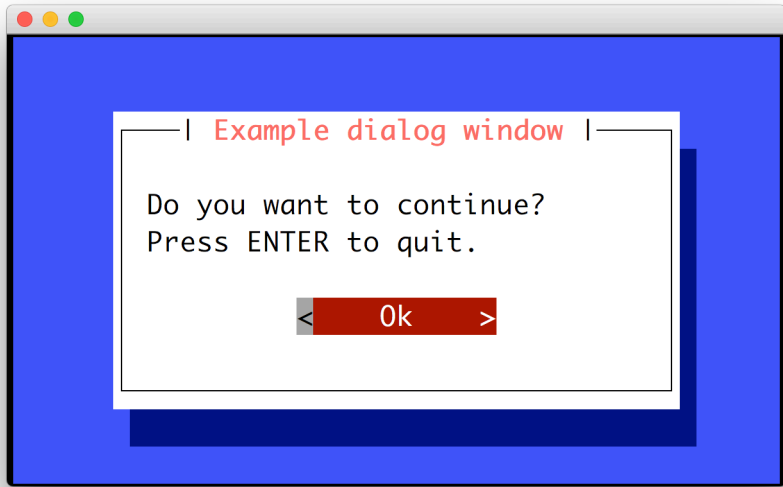
```
from prompt_toolkit.shortcuts import message_dialog

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

### Input box

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

```
from prompt_toolkit.shortcuts import input_dialog
```



**Figure 14.** Message box.

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

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

## Yes/no confirmation dialog

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

```
from prompt_toolkit.shortcuts import yes_no_dialog  
  
result = yes_no_dialog (  
    title='Yes/no dialog example',  
    text='Do you want to confirm?').run()
```

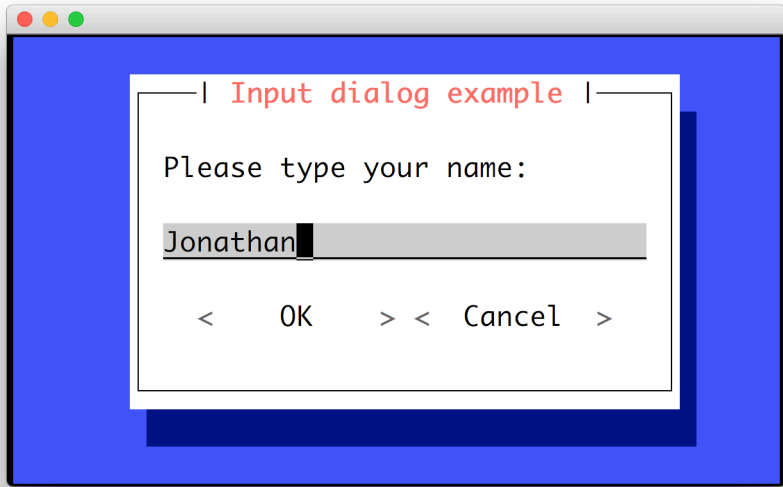


Figure 15. Input box.

## Button dialog

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

```
from prompt_toolkit.shortcuts import button_dialog

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

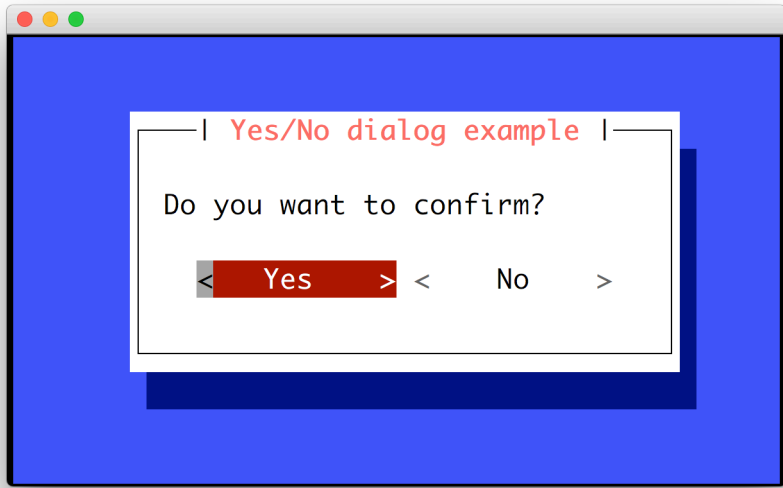


Figure 16. Yes/no confirmation dialog.

## Radio list dialog

The `prompt_toolkit.shortcuts.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).

```
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()
```

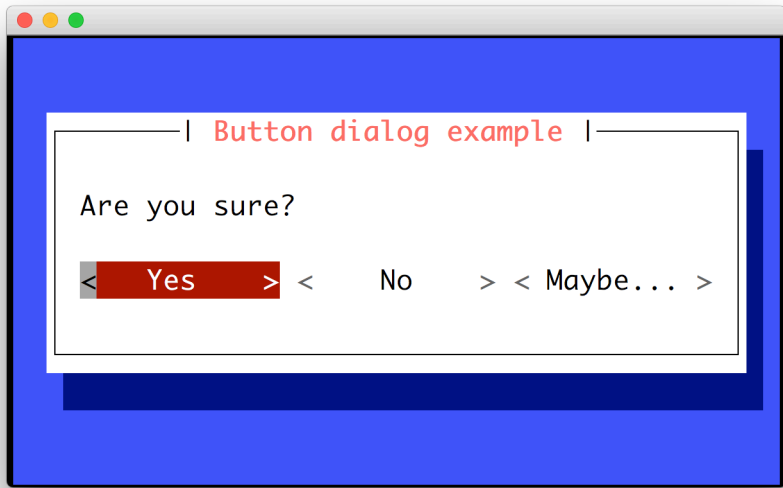


Figure 17. Button dialog.

## Checkbox list dialog

The `prompt_toolkit.shortcuts.checkboxlist_dialog` has the same usage and purpose than the Radio list dialog, but allows several values to be selected and therefore returned.

```
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()
```



Figure 18. A styled dialog window.

## Styling of dialogs

A custom `prompt_toolkit.styles.Style` instance can be passed to all dialogs to override the default style. Also, text can be styled by passing a `prompt_toolkit.formatted_text.HTML` object.

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

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

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

## 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.

Shortcut	Components used
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)
checkboxboxlist_dialog	Label CheckboxList Button (x2)
progress_dialog	Label TextArea (locked) ProgressBar

Components	Available classnames
Dialog	dialog dialog.body
TextArea	text-area text-area.prompt
Label	label
Button	button button.focused button.arrow button.text
Frame	frame frame.border frame.label
Shadow	shadow
RadioList	radio-list radio radio-checked radio-selected
CheckboxList	checkbox-list checkbox checkbox-checked checkbox-selected
VerticalLine	line vertical-line
HorizontalLine	line horizontal-line
ProgressBar	progress-bar progress-bar.used

## Example

Let's customize the example of the `checkboxlist_dialog`.

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

```
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:#cdbbb3',
        'button': 'bg:#bf99a4',
        'checkbox': '#e8612c',
        'dialog.body': 'bg:#a9cfd0',
        'dialog shadow': 'bg:#c98982',
        'frame.label': '#fcaca3',
        'dialog.body label': '#fd8bb6',
    })
).run()

```

# Chapter 6

## 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.

### Simple progress bar

Creating a new progress bar can be done by calling the `prompt_toolkit.shortcuts.ProgressBar` context manager.

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

```
# python simple-progress-bar.py
17.5% [====>                                ] 140/800 eta [00:06]
```

**Figure 19.** A simple progress bar.

```
# python two-tasks.py
49.0% [=====>] 49/100 eta [00:02]
20.7% [=====>] 31/150 eta [00:09]
```

**Figure 20.** Progress bars for multiple parallel tasks.

by the for loop and when progress happens.

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

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

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

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

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

## Multiple parallel tasks

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

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

```
from prompt_toolkit.shortcuts import ProgressBar
```

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

Figure 21. Progress bar with colored title and label.

```
import time
import threading

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

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

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

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

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

Figure 22. Modified progress bar.

## Adding a title and label

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

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

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

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

## Formatting the progress bar

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

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

default_formatting = [
    Label(),
    Text(' '),
    Percentage(),
    Text(' '),
    Bar(),
```

```

    Text(' '),
    Progress(),
    Text(' '),
    Text('eta [', style='class:time-left'),
    TimeLeft(),
    Text(']', style='class:time-left'),
    Text(' '),
]

```

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

```

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

```

```

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,

```

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

Figure 23. Custom key bindings.

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

## Adding key bindings and toolbar

Like other **Prompt toolkit** applications, we can add custom key bindings, by passing a `prompt_toolkit.key_binding.KeyBindings` object:

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

```
import 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 `prompt_toolkit.patch_stdout.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.

Read more about key bindings in chapter “*Key bindings*”.



## Chapter 7

# 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 chapter, it could be helpful to go through the chapter “*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.

## A simple application

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

```
from prompt_toolkit import Application
app = Application (full_screen=True)
app.run ()
```



## I/O objects

Every `prompt_toolkit.application.Application` instance requires an I/O object for input and output:

- A `prompt_toolkit.input.Input` instance, which is an abstraction of the input stream (*stdin*).
- A `prompt_toolkit.output.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, a `prompt_toolkit.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 appropriate handler, like for instance, a key binding.

When `prompt_toolkit.application.Application.run` is called, the event loop will run until the application is done. An application will quit when `prompt_toolkit.application.Application.exit` is called.

## 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 `prompt_toolkit.layout.Container` and `prompt_toolkit.layout.UIControl` objects.

Examples of `prompt_toolkit.layout.Container` objects are `prompt_toolkit.layout.VSplit` (vertical split), `prompt_toolkit.layout.HSplit` (horizontal split) and `prompt_toolkit.layout.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 `prompt_toolkit.layout.Window` object is a special kind of container that can contain a `prompt_toolkit.layout.UIControl` object. The `prompt_toolkit.layout.UIControl` object is responsible for the generation of the actual content. The `prompt_toolkit.layout.Window` object acts as an adaptor between the `prompt_toolkit.layout.UIControl` and other containers, but it’s also responsible for the scrolling and line wrapping of the content.

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

Normally, it is never needed to create new `prompt_toolkit.layout.UIControl` or `prompt_toolkit.layout.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 `prompt_toolkit.widgets.TextArea`, `prompt_toolkit.widgets.Button`, `prompt_toolkit.widgets.Frame`, `prompt_toolkit.widgets.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 `prompt_toolkit.layout.screen.Screen` instance.
- user controls create a `prompt_toolkit.layout.controls.UIContent` instance. This is a collection of lines that represent the actual content. A `prompt_toolkit.layout.controls.UIControl` is not aware of the screen.

## Abstract base classes, examples

`prompt_toolkit.layout.Container:`

- `prompt_toolkit.layout.HSplit`
- `prompt_toolkit.layout.VSplit`
- `prompt_toolkit.layout.FloatContainer`
- `prompt_toolkit.layout.Window`
- `prompt_toolkit.layout.ScrollablePane`

`prompt_toolkit.layout.UIControl:`

- `prompt_toolkit.layout.BufferControl`
- `prompt_toolkit.layout.FormattedTextControl`

The `prompt_toolkit.layout.Window` class itself is particular: it is a `prompt_toolkit.layout.Container` that can contain a `prompt_toolkit.layout.UIControl`. Thus, it's the adaptor between the two. The `prompt_toolkit.layout.Window` class also takes care of scrolling the content and wrapping the lines if needed.

Finally, there is the `prompt_toolkit.layout.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 `prompt_toolkit.layout.VSplit`, `prompt_toolkit.layout.HSplit` and `prompt_toolkit.layout.FloatContainer` objects.

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

Finally, there is `prompt_toolkit.layout.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 `prompt_toolkit.layout.Layout.focus` method. This method is very flexible and accepts a `prompt_toolkit.layout.Window`, a `prompt_toolkit.buffer.Buffer`, a `prompt_toolkit.layout.controls.UIControl` and more.

In the following example, we use `prompt_toolkit.application.get_app` for getting the active application.

```
from prompt_toolkit.application import get_app

# This window was created earlier:
w = Window ()
# ...
# Now focus it:
get_app().layout.focus(w)
```

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

## Key bindings

In order to react to user actions, we need to create a `prompt_toolkit.key_binding.KeyBindings` object and pass that to our `prompt_toolkit.application.Application`.

There are two kinds of key bindings:

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

### Global key bindings

Key bindings can be passed to the application as follows:

```
from prompt_toolkit import Application
```

```

from prompt_toolkit.key_binding import KeyBindings

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

```

To register a new keyboard shortcut, we can use the `prompt_toolkit.key_binding.KeyBindings.add` method as a decorator of the key handler:

```

from prompt_toolkit import Application
from prompt_toolkit.key_binding import KeyBindings

kb = KeyBindings ()

@kb.add ('c-q')
def exit_ (event):
    """
    Pressing Ctrl-Q will exit the user interface.

    Setting a return value means: quit the event loop that
    drives the user interface and return this value from
    the `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 in chapter “*Key bindings*”.

## Modal containers

The following container objects take a `modal` argument: `prompt_toolkit.layout.VSplit`, `prompt_toolkit.layout.HSplit`, and `prompt_toolkit.layout.FloatContainer`.

Setting `modal=True` makes what is called a *modal container*. Normally, a child con-



tainer would inherit its parent key bindings. This does not apply to *modal containers*.

Consider a *modal* container (e.g. `prompt_toolkit.layout.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.

## More about the Window class

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

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

A `prompt_toolkit.layout.Window` provides a “view” on the `prompt_toolkit.layout.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.

## More about buffers and BufferControl: Input processors

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

A `prompt_toolkit.layout.processors.Processor` operates on individual lines. Basically, it takes a (formatted) line and produces a new (formatted) line.

Some build-in processors of `prompt_toolkit.layout.processors`:

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

A `prompt_toolkit.layout.BufferControl` takes only one processor as input, but it is possible to “merge” multiple processors into one with the `prompt_toolkit.layout.processors.merge_processors` function.

## Chapter 8

# Tutorial: Build a SQLite REPL

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

### Read user input

Let's start accepting input using the `prompt_toolkit.shortcuts.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.

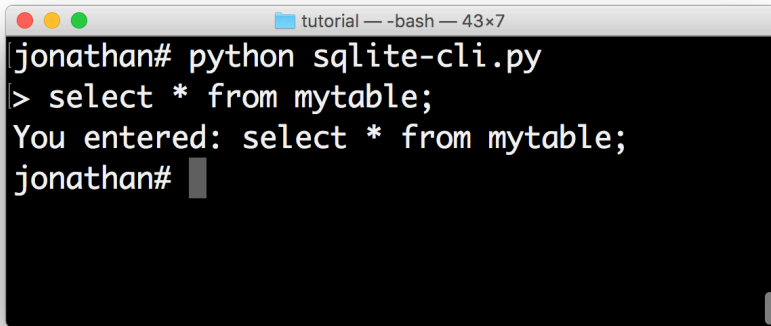
```
from prompt_toolkit import prompt

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

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

### Loop the REPL

Now we want to call the `prompt_toolkit.shortcuts.PromptSession.prompt` method in a loop. In order to keep the history, the easiest way to

A screenshot of a terminal window with a dark background. The window title bar shows three colored circles (red, yellow, green) on the left and the text 'tutorial — -bash — 43x7' on the right. The terminal content shows a prompt 'jonathan#' followed by the command 'python sqlite-cli.py'. The next line shows a prompt '>' followed by the SQL command 'select \* from mytable;'. The following line shows the text 'You entered: select \* from mytable;'. The final line shows the prompt 'jonathan#' followed by a cursor. The terminal has a scrollbar on the right side.

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

Figure 25. A simple SQLite REPL.

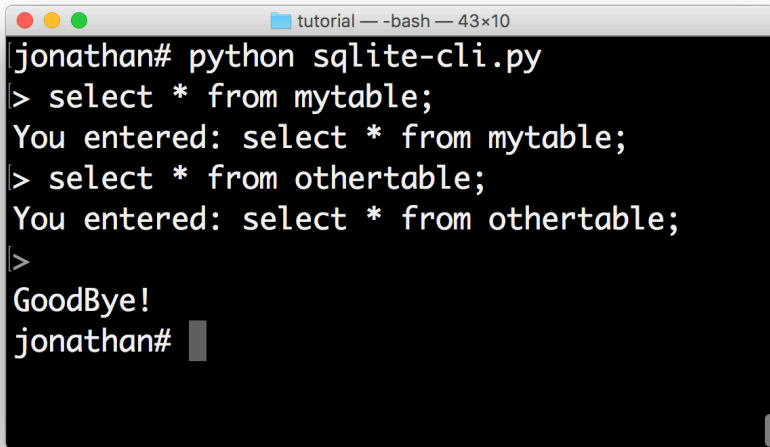
do it is to use a `prompt_toolkit.shortcuts.PromptSession`. This uses a `prompt_toolkit.history.InMemoryHistory` underneath that keeps track of the history, so that if the user presses the up arrow ↑, they'll see the previous entries.

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

```
from prompt_toolkit import PromptSession

def main ():
    session = PromptSession ()

    while True:
        try:
            text = session.prompt ('> ')
        except KeyboardInterrupt:
            continue
        except EOFError:
```

A screenshot of a terminal window with a dark background. The window title bar shows three colored circles (red, yellow, green) and the text 'tutorial — -bash — 43x10'. The terminal content shows a user prompt 'jonathan#' followed by the command 'python sqlite-cli.py'. The program then enters a loop where it prompts with '>' and displays the user's input. The first input is 'select \* from mytable;' and the second is 'select \* from othertable;'. After the second input, the program prints 'GoodBye!' and returns to the 'jonathan#' prompt. A cursor is visible after the prompt.

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

Figure 26. A SQLite REPL loop.

```
        break
    else:
        print ('You entered:', text)
    print ('GoodBye!')
```

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

## Syntax highlighting

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

Notice that in order to pass a **Pygments** lexer to **Prompt toolkit**, it needs to be

A screenshot of a terminal window with a dark background. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left and the text "tutorial — python sqlite-cli.py — 43x7" on the right. The terminal content shows a prompt "jonathan#" followed by the command "python sqlite-cli.py". Below this, a green prompt ">" is followed by the SQL query "select \* from mytable;". The text "You entered: select \* from mytable;" is displayed in white. Another green prompt ">" is shown with a cursor, indicating the next input.

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

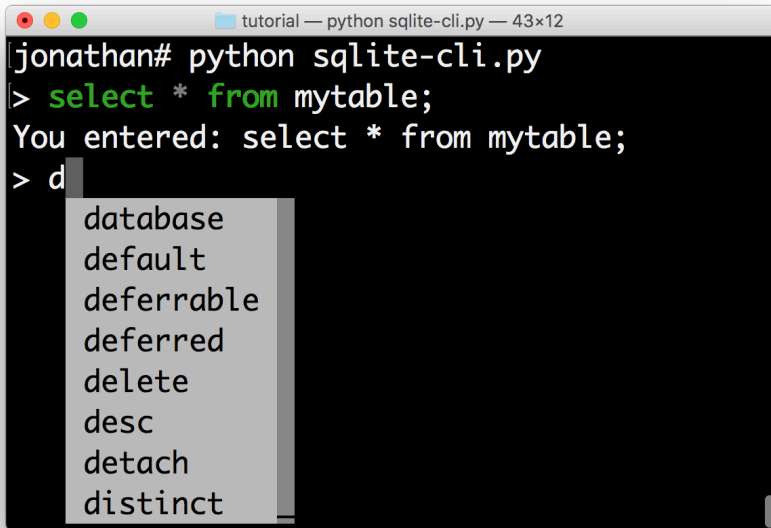
Figure 27. SQLite REPL with syntax highlighting.

wrapped into a `prompt_toolkit.lexers.PygmentsLexer`.

```
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 ()
```

A screenshot of a terminal window titled 'tutorial — python sqlite-cli.py — 43x12'. The prompt is 'jonathan#'. The user has entered 'python sqlite-cli.py'. The prompt changes to '>'. The user enters 'select \* from mytable;'. The prompt changes to 'You entered: select \* from mytable;'. The user enters 'd'. A dropdown menu appears with the following options: 'database', 'default', 'deferrable', 'deferred', 'delete', 'desc', 'detach', and 'distinct'.

```
jonathan# python sqlite-cli.py
> select * from mytable;
You entered: select * from mytable;
> d
  database
  default
  deferrable
  deferred
  delete
  desc
  detach
  distinct
```

Figure 28. SQLite REPL with autocompletion.

## Autocompletion

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

We can do this by creating a `sql_completer` object from the `prompt_toolkit.completion.WordCompleter` class, defining a set of keywords for the autocompletion.

Like the lexer, this `sql_completer` instance can be passed to either the `prompt_toolkit.shortcuts.PromptSession` class or the `prompt_toolkit.shortcuts.PromptSession.prompt` method.

```
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 (
    list_of_keywords,
    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 autocompleting, syntax highlighting REPL. Let's make it even better.

## Styling the menus

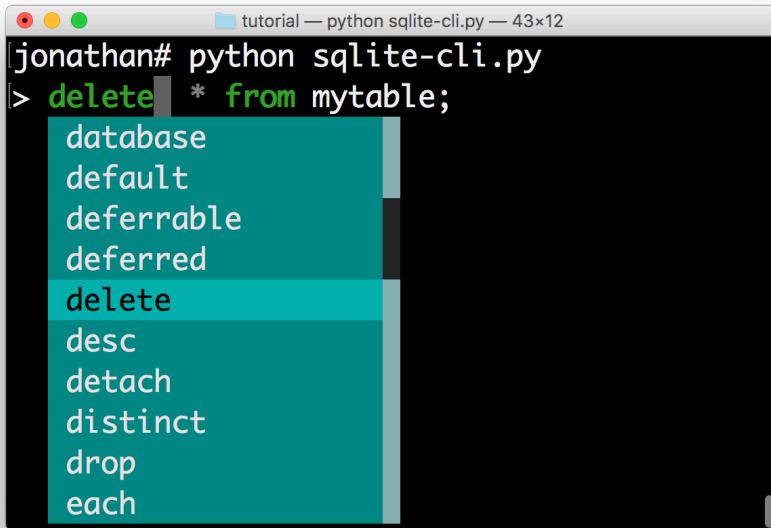
If we want, we can now change the colors of the completion menu. This is possible by creating a `prompt_toolkit.styles.Style` instance and passing it to the `prompt_toolkit.shortcuts.PromptSession.prompt` function.

```

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

```





```
jonathan# python sqlite-cli.py
> delete * from mytable;
database
default
deferrable
deferred
delete
desc
detach
distinct
drop
each
```

Figure 29. A styled SQLite REPL.

```
sql_completer = WordCompleter (
    list_of_keywords,
    ignore_case=True)

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

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

```

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

```

```

if __name__ == '__main__':
    main ()

```

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

## Hook up SQLite

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

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

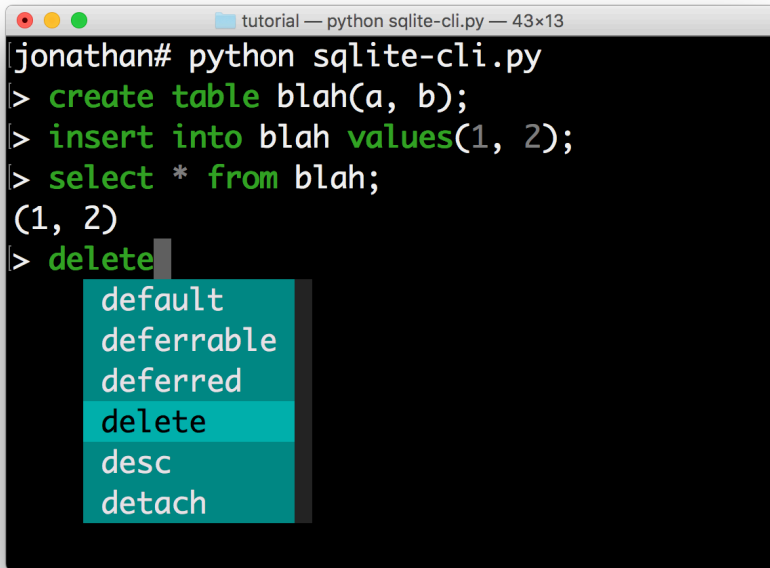
```

#!/usr/bin/env python
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 (

```



The screenshot shows a terminal window titled 'tutorial — python sqlite-cli.py — 43x13'. The prompt is 'jonathan#'. The user has entered the following SQL commands: 'python sqlite-cli.py', 'create table blah(a, b);', 'insert into blah values(1, 2);', and 'select \* from blah;'. The output of the last command is '(1, 2)'. The user has then entered 'delete' and a completion menu is displayed. The menu lists the following options: 'default', 'deferrable', 'deferred', 'delete' (which is highlighted), 'desc', and 'detach'.

```
jonathan# python sqlite-cli.py
> create table blah(a, b);
> insert into blah values(1, 2);
> select * from blah;
(1, 2)
> delete
  default
  deferrable
  deferred
  delete
  desc
  detach
```

Figure 30. A hooked up **SQLite** REPL.

```
list_of_keywords,
ignore_case=True)

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

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

```

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

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

print ('GoodBye!')

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

```

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

## Chapter 9

# More about key bindings

This chapter contains a few additional notes about key bindings.

Key bindings can be defined as follows by creating a `prompt_toolkit.key_binding.KeyBindings` instance:

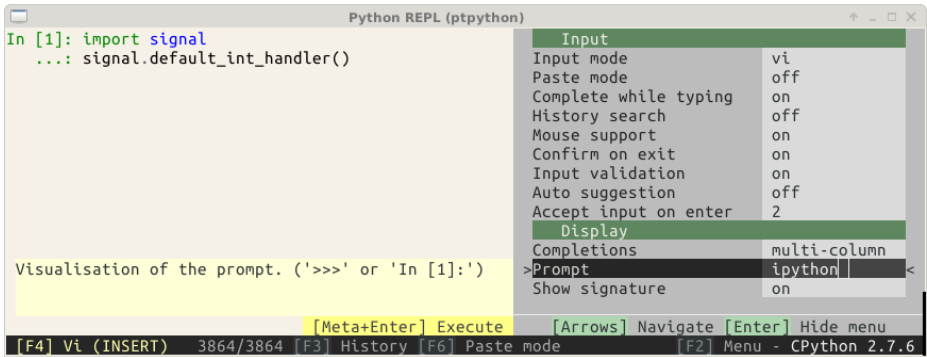
```
from prompt_toolkit.key_binding import KeyBindings

bindings = KeyBindings ()

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

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

**Note:** The `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 **Control-S** is pressed, until **Control-Q** is pressed. It won't be possible to bind these keys.



**Figure 31.** The configuration menu of **Ptpython**.

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

```
stty -ixon
```

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

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

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

## List of special keys

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

Name	Possible keys
Escape Shift +	escape s-escape
Escape	
Arrows	left, right, up, down
Navigation	home, end, delete, pageup, pagedown, insert

Name	Possible keys
Control + letter	c-a, c-b, c-c, c-d, c-e, c-f, c-g, c-h, c-i, c-j, c-k, c-l, c-m, c-n, c-o, c-p, c-q, c-r, c-s, c-t, c-u, c-v, c-w, c-x, c-y, c-z
Control + number	c-1, c-2, c-3, c-4, c-5, c-6, c-7, c-8, c-9, c-0
Control + arrow	c-left, c-right, c-up, c-down
Other Control keys	c-@, c-\, c-], c-^, c-_, c-delete
Shift + arrow	s-left, s-right, s-up, s-down
Control + Shift + arrow	c-s-left, c-s-right, c-s-up, c-s-down
Other Shift keys	s-delete, s-tab
F-keys	f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, f21, f22, f23, f24

There are a couple of useful aliases as well:

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

**Note:** The supported keys are limited to what typical VT100 terminals offer. Binding c-7 (**Control-7**) for instance is not supported.

## Binding Alt, Option or Meta key

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. Sometimes this **Alt** key is also known as **Option** or **Meta**.

In code that looks as follows:

```
@bindings.add ('escape', 'f')
```

```
def _ (event):
    " Do something if alt-f or meta-f have been pressed. "
```

## Wildcards

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

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

This will handle **AA**, **AB**, **AC**, et cetera. 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 `prompt_toolkit.filters.Filter`, usually a `prompt_toolkit.filters.Condition` instance. (Read more about filters in chapter “*Filters*”.)

```
from prompt_toolkit.filters import Condition

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

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

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



## ConditionalKeyBindings: Disabling a set of key bindings

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

```
from prompt_toolkit.key_binding import ConditionalKeyBindings

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

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

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

## Merging key bindings

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

```
from prompt_toolkit.key_binding import merge_key_bindings

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

## Eager

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

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

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

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

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

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

## Asyncio coroutines

Key binding handlers can be **Asyncio** coroutines.

```
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)
```

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.ttimeoutlen`: Like **Vim**'s `ttimeoutlen` option. When to flush the input. (For flushing escape keys.) This is important on terminals that use VT100 input. We can't distinguish the escape key from for instance the left arrow ← key, if we don't know what follows after `\x1b`. This little timer will consider `\x1b` to be escape if nothing did follow in this time span. This seems to work like the `ttimeoutlen` option in **Vim**.
- `KeyProcessor.timeoutlen`: like **Vim**'s `timeoutlen` option. This can be `None` or a float. For instance, suppose that we have a key binding **AB** and a second key binding **A**. If the 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`:

```
@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:

```
@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.

# Chapter 10

## More about styling

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

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

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

```
# Low intensity, dark:  
# (One or two components 0x80, the other 0x00.)  
ansiblack, ansired, ansigreen, ansiyellow, ansiblue,  
ansimagenta, ansicyan, ansigray  
  
# High intensity, bright:  
ansibrightblack, ansibrightred, ansibrightgreen,
```

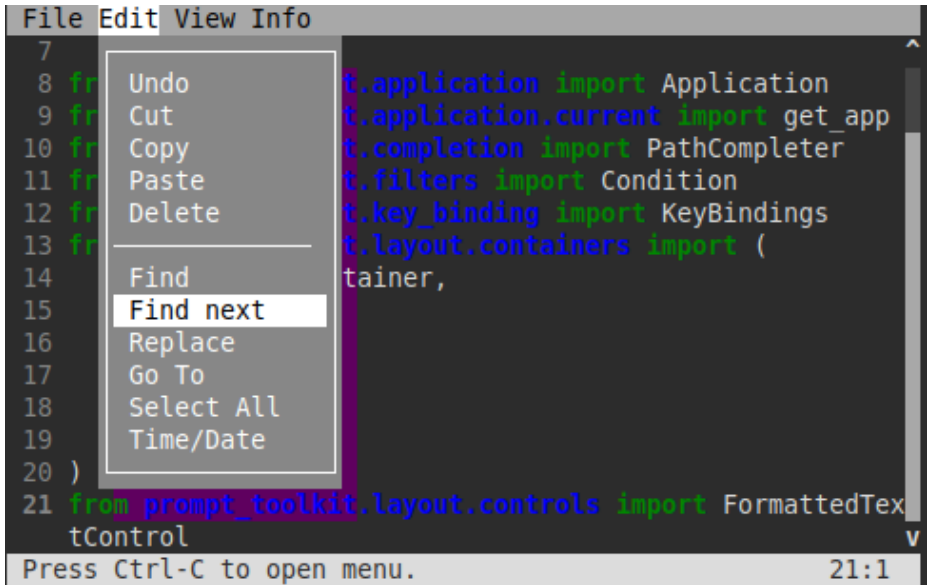


Figure 32. Text editor.

ansibrightyellow, ansibrightblue, ansibrightmagenta,  
ansibrightcyan, ansiwhite

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

## Class names

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

```
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'),
])

```

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

```

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

```

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

```

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

```

## Dot notation in class names

The dot operator has a special meaning in a class name. If we write:

```

style="class:a.b.c"

```

then this will actually expand to the following:

```

style="class:a class:a.b class:a.b.c"

```

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



## Multiple classes in a style sheet

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

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

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

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

It is possible to combine this:

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

## Evaluation order of rules in a style sheet

The style is determined as follows:

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

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

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

If we come across a class name, then we generate all combinations of the class names that we collected so far (this one and all class names to the left), and for each combination which includes the new class name, we look for

matching rules in our style sheet. All these rules are then applied (later rules have higher priority).

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

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

## Using a dictionary as a style sheet

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

```
from prompt_toolkit.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:

```
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 `prompt_toolkit.styles.Style` objects can be merged together as follows:

```
from prompt_toolkit.styles import merge_styles

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

## Color depths

There are four different levels of color depths available:

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

By default, 256 colors are used, because this is what most terminals support these days. If the `TERM` environment 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.

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

An application can also decide to set the color depth manually by passing

a `prompt_toolkit.output.ColorDepth` value to the `prompt_toolkit.application.Application` object:

```
from prompt_toolkit.output.color_depth import ColorDepth

app = Application (
    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 `prompt_toolkit.styles.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.

```
from prompt_toolkit.styles import (
    AdjustBrightnessStyleTransformation)

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

## 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 `prompt_toolkit.layout.BufferControl`;
- et cetera.

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

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

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

## An example

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

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

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

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

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

@Condition
def is_searching ():
```

```
return get_app().is_searching
```

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

```
from prompt_toolkit.key_binding import KeyBindings

kb = KeyBindings ()

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

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

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

## Built-in filters

The `prompt_toolkit.filters` has 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. These are: `has_arg`, `has_completions`, `has_focus`, `buffer_has_focus`, `has_selection`, `has_validation_error`, `is_aborting`, `is_done`, `is_read_only`, `is_multiline`, `renderer_height_is_known`, `in_editing_mode`, `in_paste_mode`, `vi_mode`, `vi_navigation_mode`, `vi_insert_mode`, `vi_insert_multiple_mode`, `vi_replace_mode`, `vi_selection_mode`, `vi_waiting_for_text_object_mode`, `vi_digraph_mode`, `emacs_mode`, `emacs_insert_mode`, `emacs_selection_mode`, `is_searching`, `control_is_searchable` and `vi_search_direction_reversed`.

## Combining filters

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

Some examples:

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

```

from prompt_toolkit.filters import (
    has_selection, has_selection)

kb = KeyBindings ()

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

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

```

## The function `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 use this value to a certain setting or event that changes over time.

In order to handle both use cases, there is a utility called `prompt_toolkit.filters.utils.to_filter`.

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

```

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

```

# Chapter 11

## The rendering flow

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

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

Let's take the following code:

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

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

What happens when a `prompt_toolkit.renderer.Renderer` objects wants a `prompt_toolkit.layout.Container` to be rendered on a certain `prompt_toolkit.layout.screen.Screen`?

The visualisation happens in several steps:



```

~/git/pymux
1  [ ] 3.9% Tasks: 130; 1 running
2  [ ] 2.0% Load average: 0.05 0.08 0.09
Mem [|||||] 553/3029MB Uptime: 17:14:01
Swp [ ] 0/4092MB

PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+
23776 jonathan 20 0 69528 20980 2204 S 2.0 0.7 0:06.77
1051 root 20 0 127M 54220 14752 S 1.3 1.7 3:22.25
24012 jonathan 20 0 203M 13800 9552 S 0.7 0.4 0:00.05
23915 jonathan 20 0 5504 1764 1312 R 0.7 0.1 0:01.05
2511 jonathan 20 0 209M 21548 11216 S 0.7 0.7 2:48.03
23627 jonathan 20 0 259M 58284 23232 S 0.0 1.9 0:03.22
2121 jonathan 20 0 126M 10616 7808 S 0.0 0.3 0:02.22
2211 jonathan 20 0 178M 15292 10924 S 0.0 0.5 0:05.87
F1help F2Setup F3Search F4Filter F5Tree F6SortBy F7Vice F8Vice

~/git/pymux
jonathan@jonathan-VirtualBox ~/git/pymux
$ ls
examples      pymux      README.rst
LICENSE       pymux.egg-info setup.py
prompt-toolkit-render-input.log README.2.rst TODO

jonathan@jonathan-VirtualBox ~/git/pymux
$ ^C

jonathan@jonathan-VirtualBox ~/git/pymux
$

commands/commands.py - Pyvim
utils.py commands.py
264 @cmd('swap-pane', options='(-D-U)')
265 def swap_pane(pymux, cli, variables):
266     pymux.arrangement.get_active_window(cli).rotate(wit
267
268
269 @cmd('kill-pane')
270 def kill_pane(pymux, cli, variables):
271     pane = pymux.arrangement.get_active_pane(cli)
272     pymux.kill_pane(pane)
273
274
275 @cmd('kill-window')
276 def kill_window(pymux, cli, variables):
277     "kill all panes in the current window."
278     for pane in pymux.arrangement.get_active_window(cli)
279         pymux.kill_pane(pane)
280
281
282 @cmd('suspend-client')
283 def suspend_client(pymux, cli, variables):
284     connection = pymux.get_connection_for_cli(cli)
285
286     if connection:
287         connection.suspend_client_to_background()
288
289
290 @cmd('clock-mode')
291 def clock_mode(pymux, cli, variables):
292     pane = pymux.arrangement.get_active_pane(cli)
293     if pane:
294         commands/commands.py (290,7) - 42%
00:39 03-Jan-16

```

Figure 33. Pymux, a terminal multiplexer (like Tmux) in Python.

1. The `prompt_toolkit.renderer.Renderer` calls the `prompt_toolkit.layout.Container.write_to_screen` method of a `prompt_toolkit.layout.Container`. This is a request to paint the layout in a rectangle of a certain size.

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

Inside the `prompt_toolkit.layout.UIControl.create_content` method of `prompt_toolkit.layout.UIControl`, there are several steps:

2. First, the buffer's text is passed to the `prompt_toolkit.lexers.Lexer.lex_document` method of a `prompt_toolkit.lexers.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 `prompt_toolkit.layout`.

`processors.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 `prompt_toolkit.layout.UIControl` returns a `prompt_toolkit.layout.UIContent` instance which generates such a token lists for each lines.

The `prompt_toolkit.layout.Window` receives the `prompt_toolkit.layout.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 `prompt_toolkit.layout.screen.Screen`, as requested by the `prompt_toolkit.renderer.Renderer`. While doing this, the `prompt_toolkit.layout.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 `prompt_toolkit.layout.Window`, then it is not requested from the `prompt_toolkit.layout.UIContent`. And from there, the line is not passed through the processors or even asked from the `prompt_toolkit.lexers.Lexer`.

## Chapter 12

# Running on top of the Asyncio event loop

**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:

```
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 ())
```

## Chapter 13

# 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 `asyncio.SelectorEventLoop`, not the `asyncio.ProactorEventLoop` (on Windows) due to the way the implementation works (contributions are welcome to make that work).

```
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.
```

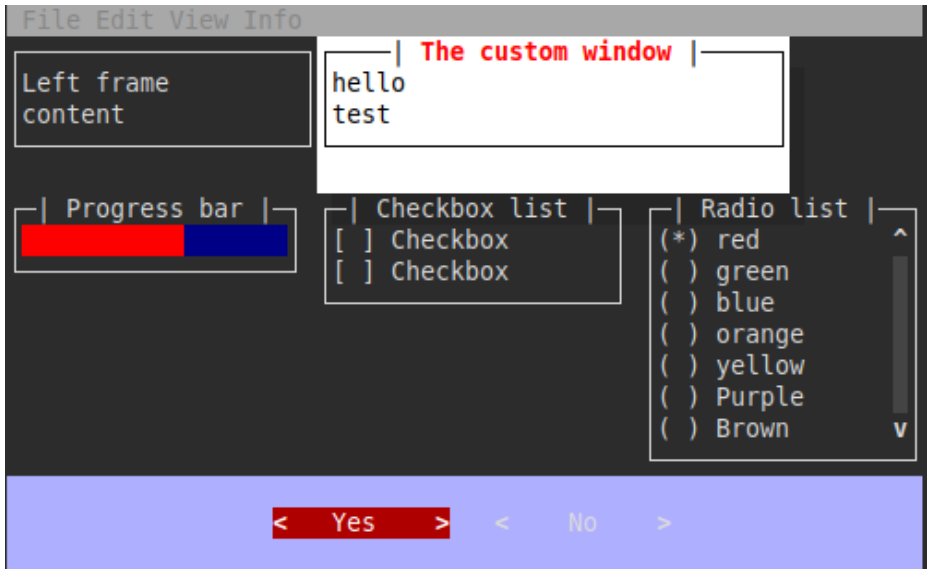


Figure 34. Full screen demo.

```
while True:
    ...

set_eventloop_with_inpthook (inpthook)

# Any asyncio code at this point will now use this new loop,
# with input hook installed.
```

# Chapter 14

## Architecture

**Note:** This is a little outdated.

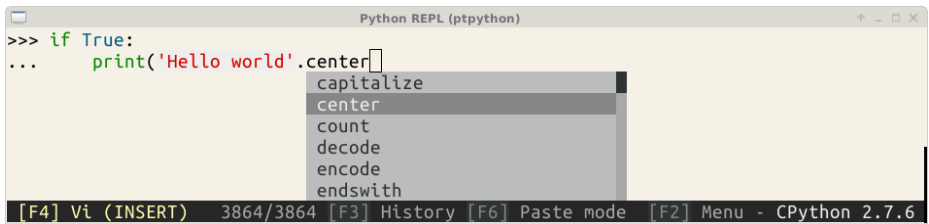
1. `InputStream`.

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`.

2. `InputStreamHandler`.

Has a `Registry` of key bindings, it calls the bindings according to the received keys and the input mode.



**Figure 35.** Ptpython, a Python REPL, the prompt.

We have **Vi** and **Emacs** bindings.

### 3. 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.

### 4. Buffer.

Contains a data structure to hold the current input (text and cursor position). This class implements all text manipulations and cursor movements (Like `cursor_forward`, `insert_char` or `delete_word`.)

### 5. Document (text, cursor\_position).

Accessed as the `document` property of the `Buffer` class. This is a wrapper around the text and cursor position, and contains methods for querying this data, e.g. to give the text before the cursor.

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

### 6. Layout.

When the renderer should redraw, the renderer asks the layout what the output should look like.

The layout operates on a `Screen` object that he received from the `Renderer` and will put the toolbars, menus, highlighted content and prompt in place.

### 7. Menus, toolbars, prompt.

### 8. Renderer.

Calculates the difference between the last output and the new one and writes it to the terminal output.

## Chapter 15

# 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 `prompt_toolkit.input.Input` object, by calling `prompt_toolkit.input.Input.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 `←` key 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 `prompt_toolkit.input.vt100_parser.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 `prompt_toolkit.key_binding.key_processor.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 in chapter “*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 `prompt_toolkit.layout.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 `prompt_toolkit.layout.Container` object to calculate its preferred height. If this is a `prompt_toolkit.layout.VSplit` or `prompt_toolkit.layout.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 `prompt_toolkit.layout.screen.Screen` object. This is like a canvas on which user controls can paint their content. The `prompt_toolkit.layout.Container.write_to_screen` method of the root `Container` is called with the screen dimensions. This will call recursively `prompt_toolkit.layout.Container.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 `prompt_toolkit.layout.Window` objects, they will do the actual painting of the `prompt_toolkit.layout.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 `prompt_toolkit.output.Output` back-end. At the end, it will position the cursor in the right place.