Pylced Documentation

Release 0.3.0a6

René Kijewski

Dec 16, 2021

CONTENTS

Python bindings for **Iced**.

Iced is a cross-platform GUI library focused on simplicity and type-safety. Inspired by Elm.

CHAPTER

ONE

INSTALLATION

```
$ pip install pyiced
```

To install from source you need to have a recent version of **Rust** installed in your $PATH.

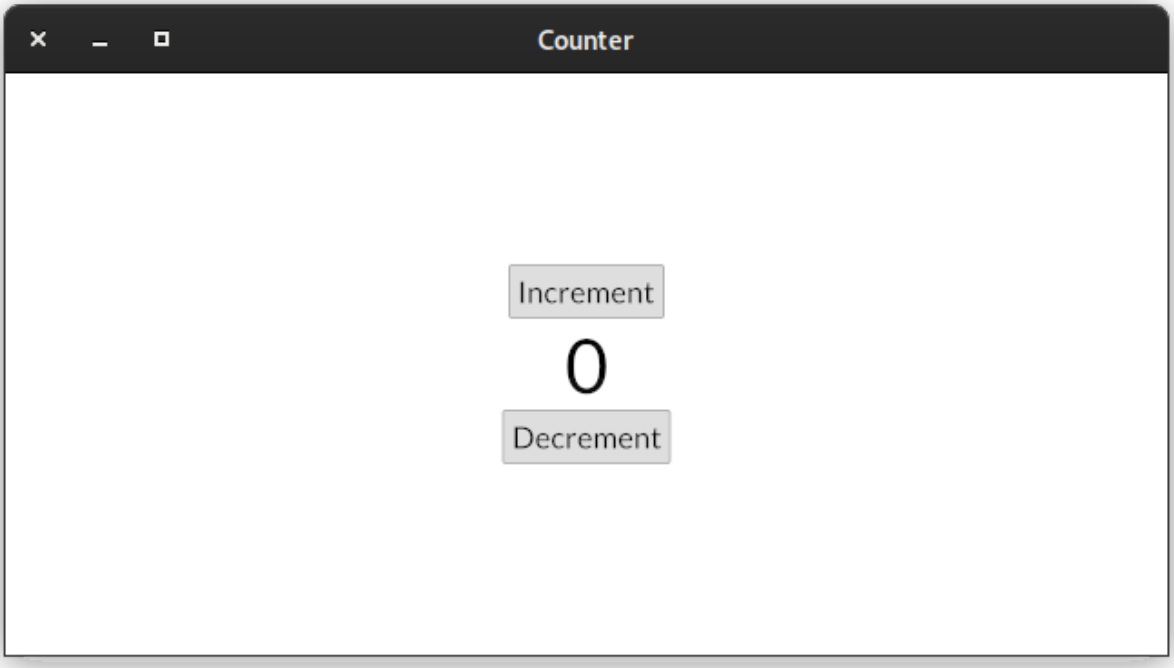
Rustup is probably the most easy to use option to install and update **Rust** on your system.

CHAPTER

TWO

QUICK EXAMPLE

A simple counter with two buttons to increment and decrement a value:




```
from pyiced import (
    Align, button, ButtonState, column, container, IcedApp, Length, text,
)


class ExampleApp(IcedApp):
    def __init__(self):
        self.__incr_button_state = ButtonState()
        self.__decr_button_state = ButtonState()
        self.__value = 0

    def title(self):
        return 'Counter'

    def view(self):
```

(continues on next page)

(continued from previous page)

```
        increment_button = button(
            self.__incr_button_state,  # To track the state across redraws.
            text('Increment'),         # This is content on the button.
            on_press='incr',           # This value is received in update().
        )
        value_label = text(f'{self.__value}', size=50)
        decrement_button = button(
            self.__decr_button_state,
            text('Decrement'),
            on_press='decr',
        )
        return container(
            column(
                [increment_button, value_label, decrement_button],
                align_items=Align.CENTER,
            ),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    def update(self, msg, clipboard):
        # When an event occurs, this method is called.
        # It can optionally return a list of async functions,
        # to handle the event.
        match msg:
            case 'incr':
                self.__value += 1
            case 'decr':
                self.__value -= 1


if __name__ == '__main__':
    # This function only returns if there is an error on start-up.
    # Otherwise the program gets terminated when the window is closed.
    ExampleApp().run()
```

CHAPTER

THREE

BIGGER EXAMPLE

Please find the source code in the *examples/chess.py*.

CHAPTER

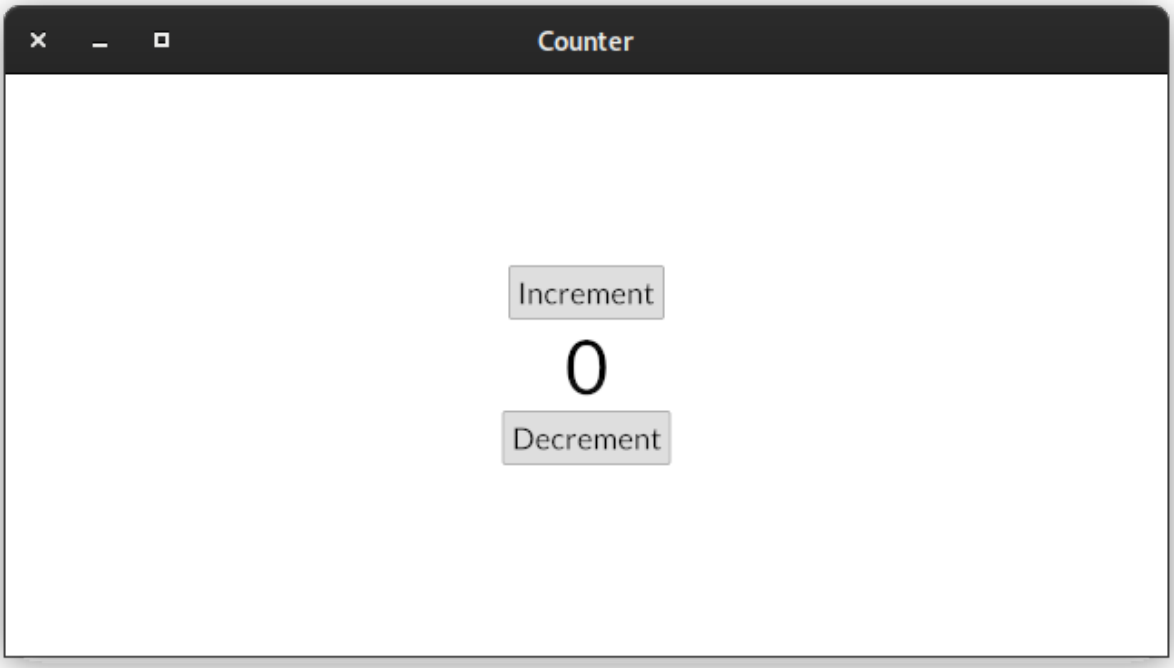
FOUR

TABLE OF CONTENTS

4.1 Usage Examples

4.1.1 Quick Example

A simple counter with two buttons to increment and decrement a value:



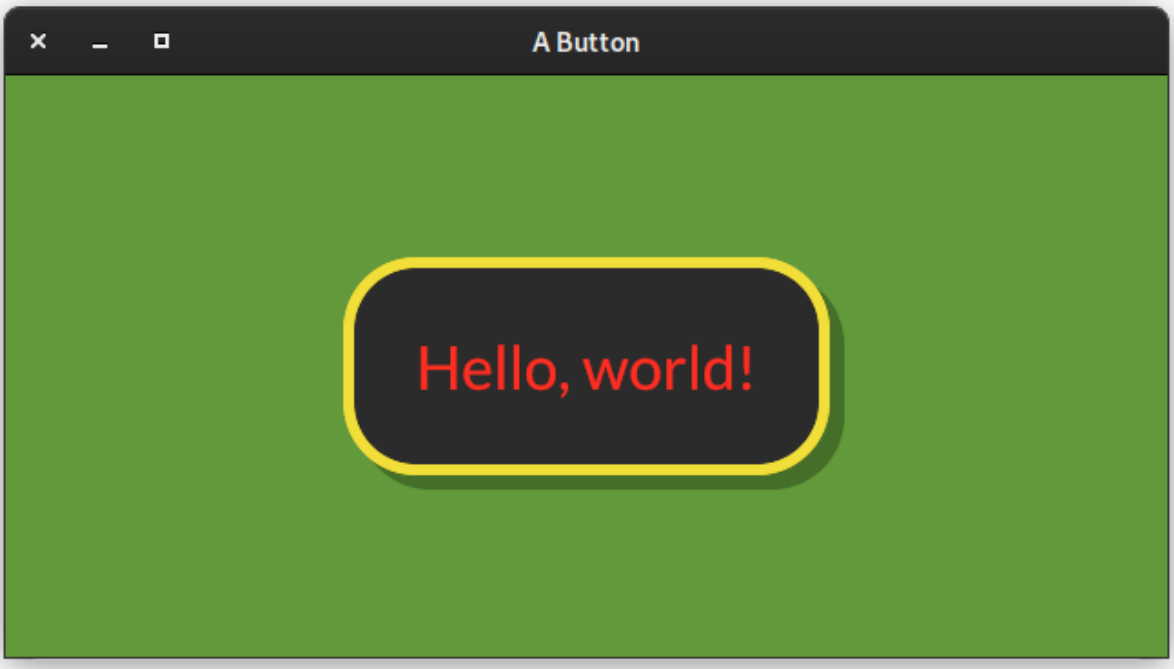

```
from pyiced import (
    Align, button, ButtonState, column, container, IcedApp, Length, text,
)


class ExampleApp(IcedApp):
    def __init__(self):
        self.__incr_button_state = ButtonState()
        self.__decr_button_state = ButtonState()
```

(continues on next page)

(continued from previous page)

```
        self.__value = 0

    def title(self):
        return 'Counter'

    def view(self):
        increment_button = button(
            self.__incr_button_state,  # To track the state across redraws.
            text('Increment'),         # This is content on the button.
            on_press='incr',           # This value is received in update().
        )
        value_label = text(f'{self.__value}', size=50)
        decrement_button = button(
            self.__decr_button_state,
            text('Decrement'),
            on_press='decr',
        )
        return container(
            column(
                [increment_button, value_label, decrement_button],
                align_items=Align.CENTER,
            ),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    def update(self, msg, clipboard):
        # When an event occurs, this method is called.
        # It can optionally return a list of async functions,
        # to handle the event.
        match msg:
            case 'incr':
                self.__value += 1
            case 'decr':
                self.__value -= 1


if __name__ == '__main__':
    # This function only returns if there is an error on start-up.
    # Otherwise the program gets terminated when the window is closed.
    ExampleApp().run()
```

4.1.2 Custom Styles




```
from pyiced import (
    Align, button, ButtonState, ButtonStyle, ButtonStyleSheet, Color,
    container, ContainerStyle, IcedApp, Length, text,
)


class ButtonExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__button_state = ButtonState()

    def title(self):
        return 'A Button'

    def view(self):
        styled_button = button(
            self.__button_state,
            text('Hello, world!', size=40),
            '',
            style=ButtonStyleSheet(ButtonStyle(
                shadow_offset=(8, 8), border_radius=40, border_width=6,
                background=Color(0.17, 0.17, 0.17),
                border_color=Color(0.95, 0.87, 0.22),
                text_color=Color(1.00, 0.18, 0.13)
```

(continues on next page)

(continued from previous page)

```
            )),
            padding=40,
        )
        return container(
            styled_button,
            style=ContainerStyle(background=Color(0.38, 0.60, 0.23)),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


if __name__ == '__main__':
    ButtonExample().run()
```

4.1.3 Asychronous Messages

new() and *update()* can either return a *Message* (or a sequence of messages in the latter case), or a coroutine / coroutines to asynchronously generate a messages.


```
from asyncio import open_connection
from contextlib import closing

from pyiced import (
    Align, Color, container, ContainerStyle, Font, IcedApp, Length, text,
)
```

(continues on next page)

(continued from previous page)

```
class AsyncMessageExample(IcedApp):
    def __init__(self):
        self.__font = None

    class settings:
        class window:
            size = (640, 320)

    def title(self):
        return 'Asynchronous Messages'

    def new(self):
        return [load_font()]

    def update(self, msg, clipboard):
        match msg:
            case ('Font', font):
                self.__font = font

    def view(self):
        return container(
            text('Hello, world!', size=80, font=self.__font),
            style=ContainerStyle(
                text_color=Color(0.95, 0.87, 0.22),
                background=Color(0.38, 0.60, 0.23),
            ),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


async def load_font():
    FONT_NAME = 'Yellowtail'
    FONT_HOST = 'fonts.cdnfonts.com'
    FONT_PATH = '/s/16054/Yellowtail-Regular.ttf'

    query = (
        f"GET {FONT_PATH} HTTP/1.0\r\n"
        f"Host: {FONT_HOST}\r\n"
        f"Connection: closed\r\n"
        f"\r\n"
    ).encode('US-ASCII')

    reader, writer = await open_connection(FONT_HOST, 443, ssl=True)
    with closing(writer):
        writer.write(query)
        await writer.drain()
        while (await reader.readline()) != b'\r\n':
            continue

        data = await reader.read()
    await writer.wait_closed()
```

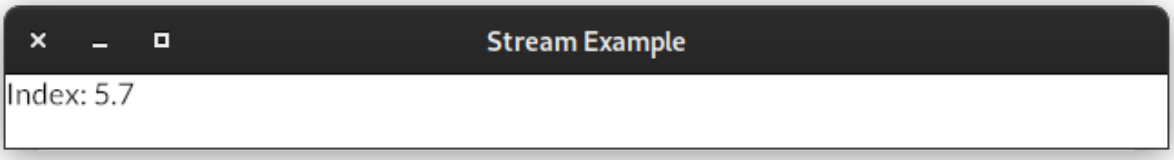
(continues on next page)

(continued from previous page)

```
    return ('Font', Font(FONT_NAME, data))


if __name__ == '__main__':
    AsyncMessageExample().run()
```

4.1.4 AsyncGenerator Generating Messages

An application can *subscribe* to AsyncGenerators to receive *Messages* about asynchronously generated information, e.g. a pending web download.




```
from asyncio import sleep

from pyiced import column, IcedApp, stream, text


class StreamExample(IcedApp):
    def __init__(self):
        self.__stream = stream(self.__generator_func())
        self.__index = 0

    class settings:
        class window:
            size = (640, 40)

    def title(self):
        return 'Stream Example'

    def view(self):
        return column([text(f'Index: {self.__index / 10:.1f}')])

    def subscriptions(self):
        if self.__stream is not None:
            return [self.__stream]

    def update(self, msg, clipboard):
        match msg:
            case 'done':
                self.__stream = None
            case int(index):
                self.__index = index
```

(continues on next page)

(continued from previous page)

```
    async def __generator_func(self):
        for i in range(1, 101):
            yield i
            await sleep(0.1)
        yield 'done'


if __name__ == '__main__':
    StreamExample().run()
```

4.1.5 Capturing Keystrokes

To capture any keystoke (or indeed any event that original from user interaction), you can make *pyiced.IcedApp.subscriptions()* return a list [*pyced.Subscription.UNCAPTURED*].

```
from pyiced import (
    Align, container, Message, IcedApp, Length, Subscription, text,
)


class FullscreenExample(IcedApp):
    def __init__(self):
        self.__fullscreen = False
        self.__should_exit = False

    class settings:
        class window:
            size = (640, 320)

    def subscriptions(self):
        return [Subscription.UNCAPTURED]

    def fullscreen(self):
        return self.__fullscreen

    def should_exit(self):
        return self.__should_exit

    def title(self):
        return self.__message

    def update(self, msg, clipboard):
        match msg:
            case Message(keyboard='keyreleased', key_code='F11'):
                self.__fullscreen = not self.__fullscreen
            case Message(keyboard='keyreleased', key_code='Escape'):
                self.__should_exit = True

    def view(self):
        return container(
            text(self.__message, size=40),
```

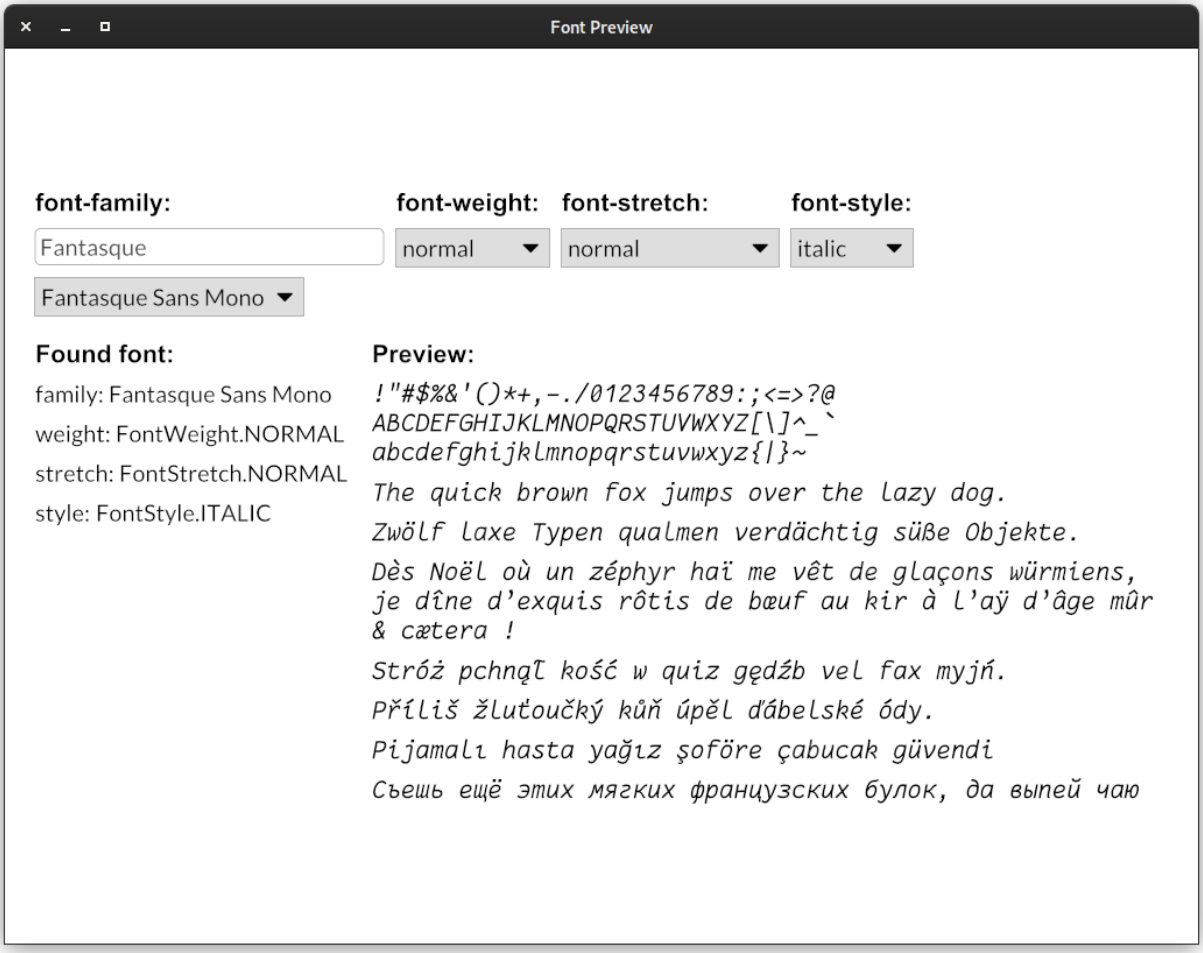
(continues on next page)

(continued from previous page)

```
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    @property
    def __message(self):
        if self.__fullscreen:
            return 'Fullscreen (press F11!)'
        else:
            return 'Windowed (press F11!)'


if __name__ == '__main__':
    FullscreenExample().run()
```

4.1.6 Using System Fonts

You can load use *findfont()* to find and load system fonts. This example gives you a preview of the installed fonts.



```
from bisect import bisect_left, bisect_right

from pyiced import (
    Align, column, container, IcedApp, Length, PickListState, pick_list,
    row, text, text_input, TextInputState, systemfonts, findfont,
)


class FontPreview(IcedApp):
    class settings:
        default_text_size = 24

    def __init__(self):
        self.__font_bold = findfont(
            ['Arial', 'Noto Sans', 'DejaVu Sans', 'sans-serif'],
            weight='bold',
        ).load()

        self.__family_prefix_state = TextInputState()
        self.__family_prefix = ''
        self.__family_state = PickListState()
        self.__family = ''
        self.__families = sorted(
            {fontid.family for fontid in systemfonts()} |
            {'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}
        )

        self.__weight = 'normal'
        self.__weight_state = PickListState()

        self.__stretch = 'normal'
        self.__stretch_state = PickListState()

        self.__style = 'normal'
        self.__style_state = PickListState()

    def title(self):
        return 'Font Preview'

    def view(self):
        if self.__family_prefix:
            def family_key(s):
                return cmp(s[:len(family_prefix)].lower(), family_prefix)

            family_prefix = self.__family_prefix.lower()
            families_start = bisect_left(
                self.__families, 0,
                key=family_key,
            )
            families_end = bisect_right(
                self.__families, 0, families_start,
                key=family_key,
            )
```

(continues on next page)

(continued from previous page)

```
            families = self.__families[families_start:families_end][:10]
        else:
            families = None
        family = column(
            [
                text('font-family:', font=self.__font_bold),
                text_input(
                    'family_prefix',
                    self.__family_prefix_state,
                    '',
                    self.__family_prefix,
                    padding=4,
                ),
                pick_list(
                    'family',
                    self.__family_state,
                    self.__family,
                    families or [
                        'serif', 'sans-serif', 'cursive', 'fantasy',
                        'monospace',
                    ],
                ),
            ],
            max_width=300,
            spacing=10,
        )
        weight = column(
            [
                text('font-weight:', font=self.__font_bold),
                pick_list(
                    'weight',
                    self.__weight_state,
                    self.__weight,
                    [
                        'thin', 'extra-light', 'light', 'normal',
                        'medium', 'semibold', 'bold', 'extra-bold',
                        'black',
                    ],
                )
            ],
            max_width=300,
            spacing=10,
        )
        stretch = column(
            [
                text('font-stretch:', font=self.__font_bold),
                pick_list(
                    'stretch',
                    self.__stretch_state,
                    self.__stretch,
                    [
                        'ultra-condensed', 'extra-condensed', 'condensed',
```

(continues on next page)

(continued from previous page)

```
                    'semi-condensed', 'normal', 'semi-expanded',
                    'expanded', 'extra-expanded', 'ultra-expanded',
                ],
            )
        ],
        max_width=300,
        spacing=10,
    )
    style = column(
        [
            text('font-style:', font=self.__font_bold),
            pick_list(
                'style',
                self.__style_state,
                self.__style,
                ['normal', 'italic', 'oblique'],
            )
        ],
        max_width=300,
        spacing=10,
    )
    search = row([family, weight, stretch, style], spacing=10)

    font = findfont(
        self.__family, self.__weight, self.__stretch, self.__style,
    )
    font_data = column(
        [
            text(
                'Found font:',
                font=self.__font_bold,
            ),
            row(
                [text('family:'), text(font.family)],
                spacing=4,
            ),
            row(
                [text('weight:'), text(repr(font.weight))],
                spacing=4,
            ),
            row(
                [text('stretch:'), text(repr(font.stretch))],
                spacing=4,
            ),
            row(
                [text('style:'), text(repr(font.style))],
                spacing=4,
            ),
        ],
        spacing=10,
    )
```

(continues on next page)

(continued from previous page)

```
        font = font.load()
        font_preview = column(
            [
                text(
                    'Preview:',
                    font=self.__font_bold,
                ),
                text(
                    '!"#$%&\'()*+,-./0123456789:;<=>?@'
                    ' ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`'
                    ' abcdefghijklmnopqrstuvwxyz{|}~',
                    font=font,
                ),
                text(
                    'The quick brown fox jumps over the lazy dog.',
                    font=font,
                ),
                text(
                    'Zwölf laxe Typen qualmen verdächtig süße Objekte.',
                    font=font,
                ),
                text(
                    'Dès Noël où un zéphyr haï me vêt de glaçons '
                    'würmiens, je dîne d’exquis rôtis de bœuf au kir '
                    'à l’aÿ d’âge mûr & cætera !',
                    font=font,
                ),
                text(
                    'Stróż pchnął kość w quiz gędźb vel fax myjń.',
                    font=font,
                ),
                text(
                    'Příliš žluťoučký kůň úpěl ďábelské ódy.',
                    font=font,
                ),
                text(
                    'Pijamalı hasta yağız şoföre çabucak güvendi',
                    font=font,
                ),
                text(
                    '      ,  '
                    ' ',
                    font=font,
                ),
            ],
            spacing=10,
        )

        return container(
            column(
                [
                    search,
```

(continues on next page)

(continued from previous page)

```
                    row(
                        [
                            font_data,
                            font_preview,
                        ],
                        spacing=20,
                    ),
                ],
                spacing=20,
            ),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    def update(self, msg, clipboard):
        match msg:
            case ('family_prefix', family_prefix):
                self.__family_prefix = family_prefix
            case ('family', family):
                self.__family = family
            case ('weight', weight):
                self.__weight = weight
            case ('stretch', stretch):
                self.__stretch = stretch
            case ('style', style):
                self.__style = style


def cmp(a, b):
    return (a > b) - (a < b)


if __name__ == '__main__':
    FontPreview().run()
```

4.1.7 Two-player Online Chess

Our last example is two-player online chess (or one player offline . . .)

It uses *subscriptions* open a TCP server / connect to a TCP server, and then await the other player's moves. It uses *commands* to tell the other player about your move.

(Please notice that this simple example does not actually know the chess rules. You can move twice, move the other player's pieces, capture your own pieces, etc.)


```
from asyncio import Future, open_connection, start_server
from contextlib import closing
from os.path import abspath, dirname, join
from traceback import print_exc

from pyiced import (
    Align, ContainerStyle, button, ButtonState, ButtonStyle,
    ButtonStyleSheet, Color, column, container, HorizontalAlignment,
    IcedApp, Length, no_element, row, stream, svg, SvgHandle, text,
    tooltip, TooltipPosition, text_input, TextInputState,
)


class ChessExample(IcedApp):
    def new(self):
        # select role:
        self.__role = None
        self.__select_role_btns = [
            ButtonState(),
            ButtonState(),
            ButtonState(),
```

(continues on next page)

(continued from previous page)

```
        ]
        self.__subscription = None

        # server role:
        self.__server_address = None

        # client role:
        self.__client_inputs = [
            TextInputState(),
            TextInputState(),
            ButtonState(),
        ]

        # playing:
        self.__writer = None
        self.__pieces = [
            [
                'Chess_tile_rd.svg',
                'Chess_tile_nd.svg',
                'Chess_tile_bd.svg',
                'Chess_tile_qd.svg',
                'Chess_tile_kd.svg',
                'Chess_tile_bd.svg',
                'Chess_tile_nd.svg',
                'Chess_tile_rd.svg',
            ],
            ['Chess_tile_pd.svg'] * 8,
            *([None] * 8 for _ in range(4)),
            ['Chess_tile_pl.svg'] * 8,
            [
                'Chess_tile_rl.svg',
                'Chess_tile_nl.svg',
                'Chess_tile_bl.svg',
                'Chess_tile_ql.svg',
                'Chess_tile_kl.svg',
                'Chess_tile_bl.svg',
                'Chess_tile_nl.svg',
                'Chess_tile_rl.svg',
            ],
        ]
        self.__pieces_root = join(
            dirname(abspath(__file__)),
            'chess-pieces',
        )
        self.__button_states = [
            [ButtonState() for _ in range(8)] for _ in range(8)
        ]
        self.__selected = None

    def title(self):
        return 'Chess Example'
```

(continues on next page)

(continued from previous page)

```
    def subscriptions(self):
        return [self.__subscription]

    def view(self):
        match self.__role:
            case 'server':
                elem = self.__view_server()
            case 'client':
                elem = self.__view_client()
            case 'playing':
                elem = self.__view_playing()
            case _:
                elem = self.__view_select_role()

        return container(
            elem,
            width=Length.FILL,
            height=Length.FILL,
            align_x=Align.CENTER,
            align_y=Align.CENTER,
        )

    def background_color(self):
        return Color(0.627, 0.612, 0.616)

    def __view_select_role(self):
        alone_state, server_state, client_state = self.__select_role_btns
        return container(
            column(
                [
                    text('Play as:'),
                    button(
                        alone_state,
                        text('Alone'),
                        ('role', 'alone'),
                        padding=4,
                    ),
                    button(
                        server_state,
                        text('Server'),
                        ('role', 'server'),
                        padding=4,
                    ),
                    button(
                        client_state,
                        text('Client'),
                        ('role', 'client'),
                        padding=4,
                    ),
                ],
                spacing=16,
                align_items=Align.CENTER,
```

(continues on next page)

(continued from previous page)

```
            ),
            style=ContainerStyle(background=Color.WHITE),
            padding=32,
        )

    def __view_server(self):
        if not self.__server_address:
            return text('Opening server ...')

        host, port = self.__server_address
        return container(
            column(
                [
                    text('Waiting for client:'),
                    text(f'Your IP: {host}'),
                    text(f'Your port: {port}'),
                ],
                spacing=16,
                align_items=Align.CENTER,
            ),
            style=ContainerStyle(background=Color.WHITE),
            padding=32,
        )

    def __view_client(self):
        if not self.__server_address:
            return text('Connecting to server ...')

        return container(
            column(
                [
                    text('Connect to server:'),
                    row(
                        [
                            text_input(
                                'host',
                                self.__client_inputs[0],
                                'Host / IP address',
                                self.__server_address[0],
                                padding=4,
                                width=Length.units(148),
                            ),
                            text_input(
                                'port',
                                self.__client_inputs[1],
                                'Port',
                                self.__server_address[1],
                                padding=4,
                                width=Length.units(148),
                            ),
                        ],
                        spacing=16,
```

(continues on next page)

(continued from previous page)

```
                ),
                button(
                    self.__client_inputs[2],
                    text(
                        'Connect',
                        horizontal_alignment=HorizontalAlignment.CENTER,
                    ),
                    ('client', self.__server_address),
                    padding=16,
                    width=Length.units(328),
                ),
            ],
            spacing=16,
            align_items=Align.CENTER,
        ),
        style=ContainerStyle(background=Color.WHITE),
        padding=32,
    )

def __view_playing(self):
    return row(
        [
            column(
                [self.__cell_at(x, y) for y in range(8)],
                width=Length.fill_portion(1),
                height=Length.FILL,
            )
            for x in range(8)
        ],
        width=Length.units(8 * 80),
        height=Length.units(8 * 80),
    )

def __cell_at(self, x, y):
    piece = self.__pieces[y][x]
    if piece:
        elem = svg(
            SvgHandle.from_path(join(self.__pieces_root, piece)),
        )
    else:
        elem = no_element()

    style = ButtonStyle(
        background=(
            Color(0.200, 0.600, 0.800)
            if self.__selected == (x, y) else
            Color(1.000, 0.808, 0.620)
            if (x + y) & 1 else
            Color(0.820, 0.545, 0.278)
        ),
        shadow_offset=(0, 0),
    )
```

(continues on next page)

(continued from previous page)

```
        return tooltip(
            button(
                self.__button_states[y][x],
                container(
                    elem,
                    align_x=Align.CENTER,
                    align_y=Align.CENTER,
                    width=Length.FILL,
                    height=Length.FILL,
                ),
                ('select', x, y, True),
                width=Length.fill_portion(1),
                height=Length.fill_portion(1),
                style=ButtonStyleSheet(style, style, style, style),
            ),
            f'{chr(ord("a") + 7 - y)}{x + 1}',
            TooltipPosition.FOLLOW_CURSOR,
        )

    def update(self, msg, clipboard):
        match msg:
            case ('select', x, y, do_notify):
                if self.__selected == (x, y):
                    # deselect
                    self.__selected = None
                elif self.__selected:
                    # move
                    (x0, y0) = self.__selected
                    self.__pieces[y][x] = self.__pieces[y0][x0]
                    self.__pieces[y0][x0] = None
                    self.__selected = None
                elif self.__pieces[y][x]:
                    # select
                    self.__selected = (x, y)

                if do_notify and self.__writer:
                    async def write():
                        self.__writer.write(b'%d %d\n' % (x, y))
                        await self.__writer.drain()
                    return [write()]

            case ('role', 'alone'):
                self.__role = 'playing'

            case ('role', 'server'):
                self.__role = 'server'
                self.__subscription = stream(self.__role_server())

            case ('role', 'client'):
                self.__role = 'client'
                self.__server_address = ['0.0.0.0', '']
```

(continues on next page)

(continued from previous page)

```
            case ('server', (host, port)):
                self.__server_address = host, port

            case ('client', (host, port)):
                self.__server_address = None
                self.__role = 'server'
                self.__subscription = stream(self.__role_client(host, port))

            case ('connected', (reader, writer)):
                self.__writer = writer
                self.__subscription = stream(self.__read_connection(reader))
                self.__role = 'playing'

            case ('host', value):
                self.__server_address[0] = value

            case ('port', value):
                self.__server_address[1] = value

            case ('host' | 'port', None, 'submit'):
                return [('client', self.__server_address)]

    async def __read_connection(self, reader):
        while not reader.at_eof():
            line = await reader.readline()
            if not line:
                break
            x, y = line.split()
            yield 'select', int(x), int(y), False

    async def __role_client(self, host, port):
        try:
            yield 'connected', await open_connection(host, port)
        except Exception:
            print_exc()
            yield 'role', 'client'

    async def __role_server(self):
        query = (
            b'GET / HTTP/1.0\r\n'
            b'Host: whatismyip.akamai.com\r\n'
            b'Connection: closed\r\n'
            b'\r\n'
        )
        reader, writer = await open_connection('whatismyip.akamai.com', 80)
        with closing(writer):
            writer.write(query)
            await writer.drain()
            while (await reader.readline()) != b'\r\n':
                continue
            hostname = (await reader.read()).decode('US-ASCII').strip()
        await writer.wait_closed()
```

(continues on next page)

(continued from previous page)

```
        client = Future()
        server = await start_server(
            lambda reader, writer: client.set_result((reader, writer)),
            '0.0.0.0',
            0,
        )
        port = next(iter(server.sockets)).getsockname()[1]
        yield 'server', (hostname, port)
        yield 'connected', await client


if __name__ == '__main__':
    ChessExample().run()
```

4.2 Programming an IcedApp

4.2.1 Overview

IcedApp()	TODO
Element	A displayable widget that can be used in *view()*.
Message	A message generated through user interaction.
Settings()	TODO
WindowSettings()	TODO

4.2.2 Details

class pyiced.**IcedApp**

TODO

background_color()

Returns the background color of the application. Defaults to white.

Return type Optional[*Color*, None]

fullscreen()

True if the program should run in fullscreen mode.

The runtime will automatically transition your application if a new mode is returned.

Return type bool

new()

Initialize the application.

You can return *Commands* if you need to perform some async action in the background on startup. This is useful if you want to load state from a file, perform an initial HTTP request, etc.

Return type Optional[Iterable[Union[Awaitable[Optional[object, None]], object, None]], None]

run(*, *run=None*)
Runs the application.

This method will take control of the current thread and will NOT return unless there is an error during startup.

It should probably be that last thing you call in your main function.

Parameters **run** (Optional[Callable[[Awaitable[Any]], Union[None, Any, NoReturn]], None]) – Coroutine executor. Defaults to asyncio.run().

Return type NoReturn

scale_factor()
Returns the scale factor of the application.

It can be used to dynamically control the size of the UI at runtime (i.e. zooming).

For instance, a scale factor of 2.0 will make widgets twice as big, while a scale factor of 0.5 will shrink them to half their size.

Return type float

property settings: Optional[*pyiced.Settings*]
The initial settings of the program.

Only queried once.

Return type Optional[*Settings*, None]

should_exit()
Returns whether the application should be terminated.

This will kill the Python instance, too.

Return type bool

subscriptions()
Returns the event *subscriptions* for the current state of the application.

A subscription will be kept alive as long as you keep returning it, and the messages produced will be handled by update.

Return type Optional[Iterable[Optional[*Subscription*, None]], None]

title()
The current title of the application.

This title can be dynamic! The runtime will automatically update the title of your application when necessary.

Return type str

update(*msg*, *clipboard*)
Handles a message and updates the state of the application.

This is where you define your update logic. All the messages, produced by either user interactions or commands, will be handled by this method.

Any *Command* returned will be executed immediately in the background.

Return type Optional[Iterable[Union[Awaitable[Optional[object, None]], object, None]], None]

abstract view()
Returns the *widget* to display in the application.

These widgets can produce messages based on user interaction.

Return type *Element*

class pyiced.**Element**
A displayable widget that can be used in *view()*.

class pyiced.**Message**
A message generated through user interaction.

Messages get passed to to *update()*.

alt
The alt key was pressed / released.

Returns

- *True* – Yes, the alt key was pressed or released.
- *False* – No, the state of the alt key is unchanged.
- *None* – Not a *"keypress"*, *"keyrelease"* or *"modifierschanged"* event.

Return type Optional[bool]

amount
The scroll movement.

Returns

The horizontal and vertical amount. The unit can be determined by inspecting *wheelscrolled*.

None if not a scroll movement.

Return type Optional[Tuple[float, float]]

button
The button of a mouse event.

Returns

- *"left"* – The left mouse button.
- *"right"* – The right mouse button.
- *"middle"* – The middle (wheel) button.
- int – Another button.
- *None* – Not a mouse event.

Return type Union[str|int|None]

characterreceived
A unicode character was received.

Returns The received, composed character. *None* if not a character event.

Return type Optional[str]

control
The control key was pressed / released.

Returns

- *True* – Yes, the control key was pressed or released.
- *False* – No, the state of the control key is unchanged.

- *None* – Not a *"keypress"*, *"keyrelease"* or *"modifierschanged"* event.

Return type Optional[bool]

`cursormoved`
The mouse cursor was moved.

Returns Horizontal and vertical pixels, or *None* if cursor was not moved.

Return type Optional[Tuple[float, float]]

`file`
The path of the hovering or dropped file.

Returns The path or none, if the Message is not a file action.

Return type Optional[pathlib.Path]

`finger`
A unique identifier representing a finger on a touch interaction.

Returns Identifier of the finger.

Return type int

`key_code`
The name of the pressed or released key.

See iced_native::keyboard::KeyCode for the name of the keys.

Returns The name of the key, or *None* if the not a key press or release.

Return type Optional[str]

`keyboard`
The kind of the keyboard interaction.

Returns

- *None* if not a `Message(native="keyboard")` interaction
- *"keypressed"* when a key was pushed down
- *"keyreleased"* when a key no more pushed down
- *"characterreceived"* when a key press + release generated a character
- *"modifierschanged"* when a modifier was pressed or released, e.g. shift

Return type Optional[str]

`kind`
The kind of the native message.

Returns

- *"mouse"* for mouse interactions, e.g. mouse clicking
- *"window"* for window interactions, e.g. resizing
- *"keyboard"* for keyboard interactions, e.g. key presses
- *"touch"* for touch interactions (impossible?)

Return type str

`logo`
The "logo" key was pressed / released.

The logo key is the windows key, command key, etc.

Returns

- *True* – Yes, the "logo" key was pressed or released.
- *False* – No, the state of the "logo" key is unchanged.
- *None* – Not a *"keypress"*, *"keyrelease"* or *"modifierschanged"* event.

Return type Optional[bool]

`mouse`
A mouse event.

Returns

- *"cursorentered"* – The mouse cursor entered the window.
- *"cursorleft"* – The mouse cursor left the window.
- *"cursormoved"* – The mouse cursor was moved
- *"buttonpressed"* – A mouse button was pressed.
- *"buttonreleased"* – A mouse button was released.
- *"wheelscrolled"* – The mouse wheel was scrolled.
- *None* – Not a mouse event.

Return type Optional[str]

`position`
A 2D point for the touch interaction.

Returns A 2D point

Return type Tuple[float, float]

`resized`
The new size of the window.

Returns The width and height in pixels or null, if it's not a resize action.

Return type Optional[tuple(int, int)]

`shift`
The shift key was pressed / released.

Returns

- *True* – Yes, the shift key was pressed or released.
- *False* – No, the state of the shift key is unchanged.
- *None* – Not a *"keypress"*, *"keyrelease"* or *"modifierschanged"* event.

Return type Optional[bool]

`touch`
A touch interaction.

Returns

- *"fingerpressed"* – A touch interaction was started.

- *"fingermoved"* – An on-going touch interaction was moved.
- *"fingerlifted"* – A touch interaction was ended.
- *"fingerlost"* – A touch interaction was canceled.
- *None* – Not a touch interaction.

Return type Optional[str]

wheelscrolled
The unit of the scroll movement.

Returns

- *"lines"* – Counting in lines / columns.
- *"pixels"* – Counting in pixels.
- *None* – Not a scroll movement.

Return type Optional[str]

window
The kind of the window message.

Returns

- *"resized"* – The window was resized.
- *"closerequested"* – The window close button was clicked.
- *"focused"* – The window was focus.
- *"unfocused"* – The focus left the window.
- *"filehovered"* – A file is hovering the window. A selection of multiple files cause multiple messages.
- *"filedropped"* – A file was dropped on the window. A selection of multiple files cause multiple messages.
- *"fileshoveredleft"* – The cursor the with hovering file(s) left the window.
- *None* – Not a window message.

Return type Optional[str]

class pyiced.**Settings**
TODO

property antialiasing: bool
If set to true, the renderer will try to perform antialiasing for some primitives.

Enabling it can produce a smoother result in some widgets, like the Canvas, at a performance cost.

Return type bool

property default_font: Optional[*pyiced.Font*]
The font that will be used by default.

If *None* or *Font.DEFAULT* is provided, a default system font will be chosen.

Return type Optional[*Font*, None]

property default_text_size: int
The text size that will be used by default.

Return type int

property exit_on_close_request: bool
Whether the *IcedApp* should exit when the user requests the window to close (e.g. the user presses the close button).

Return type bool

property window: Optional[*pyiced.WindowSettings*]
The window settings.

Return type Optional[*WindowSettings*, None]

class pyiced.**WindowSettings**
TODO

property always_on_top: bool
Whether the window will always be on top of other windows.

Return type bool

property decorations: bool
Whether the window should have a border, a title bar, etc. or not.

Return type bool

property max_size: Optional[Tuple[int, int]]
The maximum size of the window.

Return type Optional[Tuple[int, int], None]

property min_size: Optional[Tuple[int, int]]
The minimum size of the window.

Return type Optional[Tuple[int, int], None]

property resizable: bool
Whether the window should be resizable or not.

Return type bool

property size: Tuple[int, int]
Dimensions of the newly crated window.

Return type Tuple[int, int]

property transparent: bool
Whether the window should be transparent.

Return type bool

4.2.3 Type aliases

pyiced.**Command**
alias of Union[Awaitable[Optional[object]], object]

pyiced.**Commands**
alias of Iterable[Optional[Union[Awaitable[Optional[object]], object]]]

4.3 Displayable Elements

4.3.1 Overview

button(state, content[, on_press, width, ...])	A generic widget that produces a message when pressed.
checkbox(token, is_checked, label, *[, ...])	A box that can be checked.
column(children, *[, spacing, padding, ...])	A container that distributes its contents vertically.
container(content, *[, padding, width, ...])	An element decorating some content.
image(handle, *[, width, height])	A frame that displays an image while keeping aspect ratio.
no_element()	A *space()* with minimum width and height.
pick_list(token, state, selected, options, *)	A widget for selecting a single value from a list of options.
progress_bar(start, end, value, *[, width, ...])	A bar that displays progress.
radio(token, selected, value, label, *[, ...])	A circular button representing a choice.
row(children, *[, spacing, padding, width, ...])	A container that distributes its contents horizontally.
rule(*[, horizontal, vertical, style])	Display a horizontal or vertical rule for dividing content.
scrollable(state, children, *[, spacing, ...])	A widget that can vertically display an infinite amount of content with a scrollbar.
slider(token, state, start, end, value[, ...])	An horizontal bar and a handle that selects a single value from a range of values.
space(*[, width, height])	An amount of empty space.
svg(handle, *[, width, height])	A vector graphics image.
text(label, *[, size, color, font, width, ...])	A paragraph of text.
text_input(token, state, placeholder, value, *)	A field that can be filled with text.
tooltip(content, tooltip, position, *[, ...])	Make a tooltip.

4.3.2 Details

pyiced.**button**(*state*, *content*, *on_press=None*, ***, *width=None*, *height=None*, *min_width=None*, *min_height=None*, *padding=None*, *style=None*)

A generic widget that produces a message when pressed.

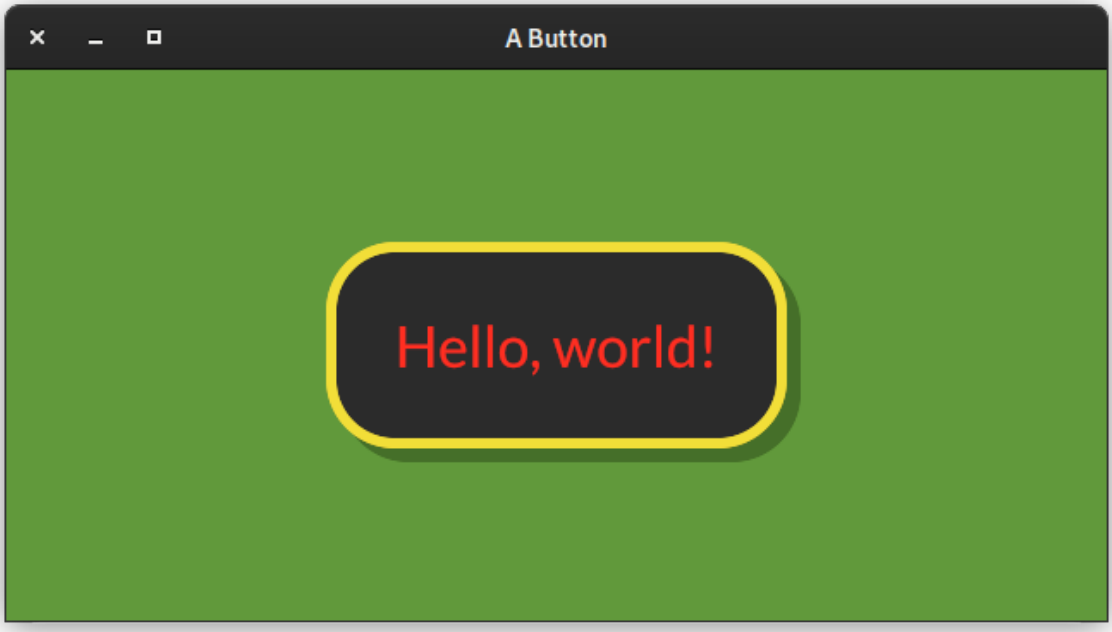
Parameters

- **state** (ButtonState) – Current state of the button. The same object must be given between calls.
- **content** (Element) – The element displayed inside the button, e.g. a *text()*.
- **on_press** (*Optional[object]*) – Message to send to the app's *update()* loop when the key was clicked. Without this argument the button won't be clickable.
- **width** (*Optional[Length]*) – Width the the button.
- **height** (*Optional[Length]*) – Height the the button.
- **min_width** (*Optional[int]*) – Minimum width of the button in pixels.
- **min_height** (*Optional[int]*) – Minimum height of the button in pixels.
- **padding** (*Optional[int]*) – Amount of pixels surrounding the contained element.
- **style** (*Optional[ButtonStyleSheet]*) – The style of the button.

Returns The newly created button.

Return type *Element*

Example




```
from pyiced import (
    Align, button, ButtonState, ButtonStyle, ButtonStyleSheet, Color,
    container, ContainerStyle, IcedApp, Length, text,
)


class ButtonExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__button_state = ButtonState()

    def title(self):
        return 'A Button'

    def view(self):
        styled_button = button(
            self.__button_state,
            text('Hello, world!', size=40),
            '',
            style=ButtonStyleSheet(ButtonStyle(
                shadow_offset=(8, 8), border_radius=40, border_width=6,
                background=Color(0.17, 0.17, 0.17),
```

(continues on next page)

(continued from previous page)

```
                border_color=Color(0.95, 0.87, 0.22),
                text_color=Color(1.00, 0.18, 0.13)
            )),
            padding=40,
        )
        return container(
            styled_button,
            style=ContainerStyle(background=Color(0.38, 0.60, 0.23)),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


if __name__ == '__main__':
    ButtonExample().run()
```

See also:

iced_native::widget::button::Button

pyiced.**checkbox**(*token*, *is_checked*, *label*, *, *size=None*, *width=None*, *spacing=None*, *text_size=None*, *font=None*, *style=None*)

A box that can be checked.

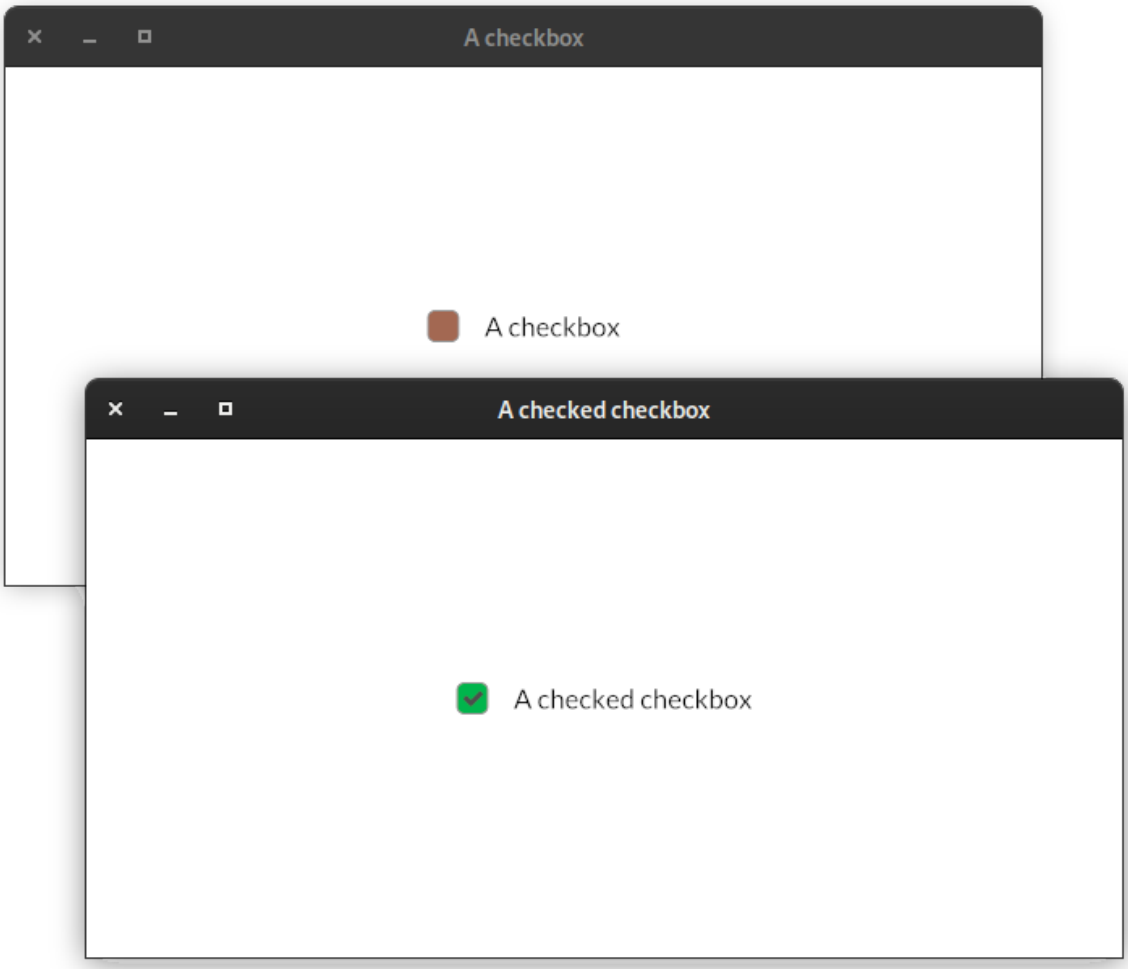
Parameters

- **token** (*object*) – When the user changes the text, a message `(token, new_is_checked)` is sent to the app's *update()* method.
- **is_checked** (*bool*) – Whether the checkbox is currently checked or not.
- **label** (*str*) – A text besides the checkbox. Might be empty.
- **size** (*Optional[int]*) – Size of the checkbox.
- **width** (*Optional[Length]*) – Width of the widget (checkbox and text).
- **spacing** (*Optional[int]*) – Space between checkbox and text.
- **text_size** (*Optional[int]*) – Font size of the text.
- **font** (*Optional[Font]*) – Font of the text.
- **style** (*Optional[CheckboxStyleSheet]*) – Style of the checkbox.

Returns Newly created checkbox.

Return type *Element*

Example




```
from pyiced import (
    Align, checkbox, CheckboxStyle, CheckboxStyleSheet, Color, container,
    IcedApp, Length,
)


class CheckboxExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__is_checked = False

    def title(self):
        if self.__is_checked:
            return 'A checked checkbox'
        else:
```

(continues on next page)

(continued from previous page)

```
            return 'A checkbox'

    def view(self):
        styled_checkbox = checkbox(
            'set',
            self.__is_checked,
            self.title(),
            style=CheckboxStyleSheet(
                active=CheckboxStyle(
                    'active',
                    background=Color(0.64, 0.41, 0.32),
                ),
                active_checked=CheckboxStyle(
                    'active_checked',
                    background=Color(0, 0.71, 0.296),
                ),
            ),
        )
        return container(
            styled_checkbox,
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    def update(self, msg, clipboard):
        match msg:
            case 'set', is_checked:
                self.__is_checked = is_checked


if __name__ == '__main__':
    CheckboxExample().run()
```

See also:

iced_native::widget::checkbox::Checkbox

pyiced.**column**(*children*, ***, *spacing=None*, *padding=None*, *width=None*, *height=None*, *max_width=None*, *max_height=None*, *align_items=None*)

A container that distributes its contents vertically.

Parameters

- **children** (*Iterable[Optional[Element]]*) – Create the column with the given elements.
- **spacing** (*Optional[int]*) – Vertical spacing between elements.
- **padding** (*Optional[int]*) – Padding of the column.
- **width** (*Optional[Length]*) – Width of the column.
- **height** (*Optional[Length]*) – Height of the column.
- **max_width** (*Optional[int]*) – Maximum width of the column.
- **max_height** (*Optional[int]*) – Maximum height of the column in pixels.

- **align_items** (*Optional[Align]*) – Horizontal alignment of the contents of the column.

Returns The newly created column.

Return type *Element*

Example


```
from pyiced import column, IcedApp, text


class ColumnExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def title(self):
        return 'A Column'

    def view(self):
        return column(
            [text('Hello,'), text('world!')],
            padding=80, spacing=120,
        )


if __name__ == '__main__':
    ColumnExample().run()
```

See also:

iced_native::widget::column::Column

pyiced.**container**(*content*, ***, *padding=None*, *width=None*, *height=None*, *max_width=None*, *max_height=None*, *align_x=None*, *align_y=None*, *style=None*)

An element decorating some content.

It is normally used for alignment purposes.

Parameters

- **content** (Element) – The content of the container.
- **padding** (*Optional[int]*) – The padding around the content.
- **width** (*Optional[Length]*) – The width of the container.
- **height** (*Optional[Length]*) – The height of the container.
- **max_width** (*Optional[int]*) – The maximum width of the container
- **max_height** (*Optional[int]*) – The maximum height of the container.
- **align_x** (*Optional[Length]*) – The horizontal alignment of the content inside the container.
- **align_y** (*Optional[Length]*) – The vertical alignment of the content inside the container.
- **style** (*Optional[ContainerStyleSheet]*) – The style of the container.

Returns The newly created .

Return type *Element*

See also:

iced_native::widget::container::Container

pyiced.**image**(*handle*, ***, *width=None*, *height=None*)

A frame that displays an image while keeping aspect ratio.

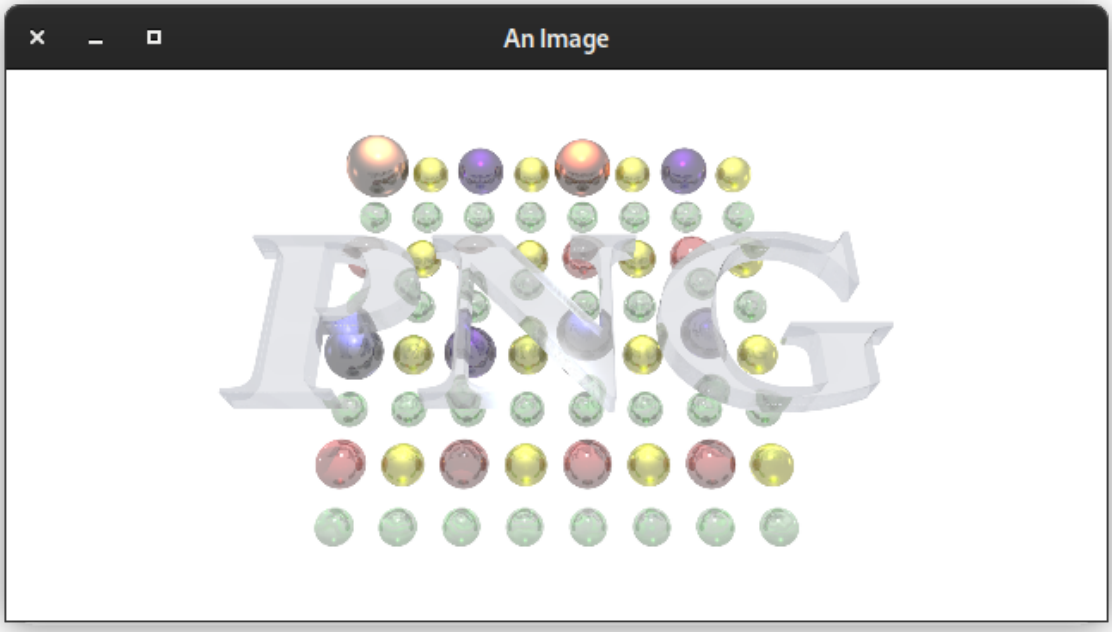
Parameters

- **handle** (ImageHandle) – The handle of the image.
- **width** (*Optional[Length]*) – The width of the image.
- **heigth** (*Optional[Length]*) – The height of the image.

Returns The newly created image element.

Return type *Element*

Example




```
from asyncio import open_connection
from contextlib import closing

from pyiced import (
    Align, container, IcedApp, image, ImageHandle, Length, text,
)


class ImageExample(IcedApp):
    def __init__(self):
        self.__handle = None

    class settings:
        class window:
            size = (640, 320)

    def title(self):
        return 'An Image'

    def new(self):
        return [load_image()]

    def update(self, msg, clipboard):
        match msg:
            case ('ImageHandle', handle):
                self.__handle = handle

    def view(self):
```

(continues on next page)

(continued from previous page)

```
        if self.__handle is None:
            return text('Loading ...')

        return container(
            image(
                self.__handle,
                height=Length.units(300),
                width=Length.units(600),  # the aspect ratio is preserved
            ),
            align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


async def load_image():
    HOST = 'upload.wikimedia.org'
    PATH = '/wikipedia/de/b/bb/Png-logo.png'

    query = (
        f"GET {PATH} HTTP/1.0\r\n"
        f"Host: {HOST}\r\n"
        f"Connection: closed\r\n"
        f"User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)\r\n"
        f"\r\n"
    ).encode('US-ASCII')

    reader, writer = await open_connection(HOST, 443, ssl=True)
    with closing(writer):
        writer.write(query)
        await writer.drain()
        while (await reader.readline()) != b'\r\n':
            continue

        data = await reader.read()
    await writer.wait_closed()

    return ('ImageHandle', ImageHandle.from_memory(data))


if __name__ == '__main__':
    ImageExample().run()
```

See also:

iced_native::widget::image::Image

pyiced.**no_element**()

A *space()* with minimum width and height.

Returns The newly created empty space.

Return type *Element*

pyiced.**pick_list**(*token*, *state*, *selected*, *options*, ***, *text_size=None*, *font=None*, *style=None*)

A widget for selecting a single value from a list of options.

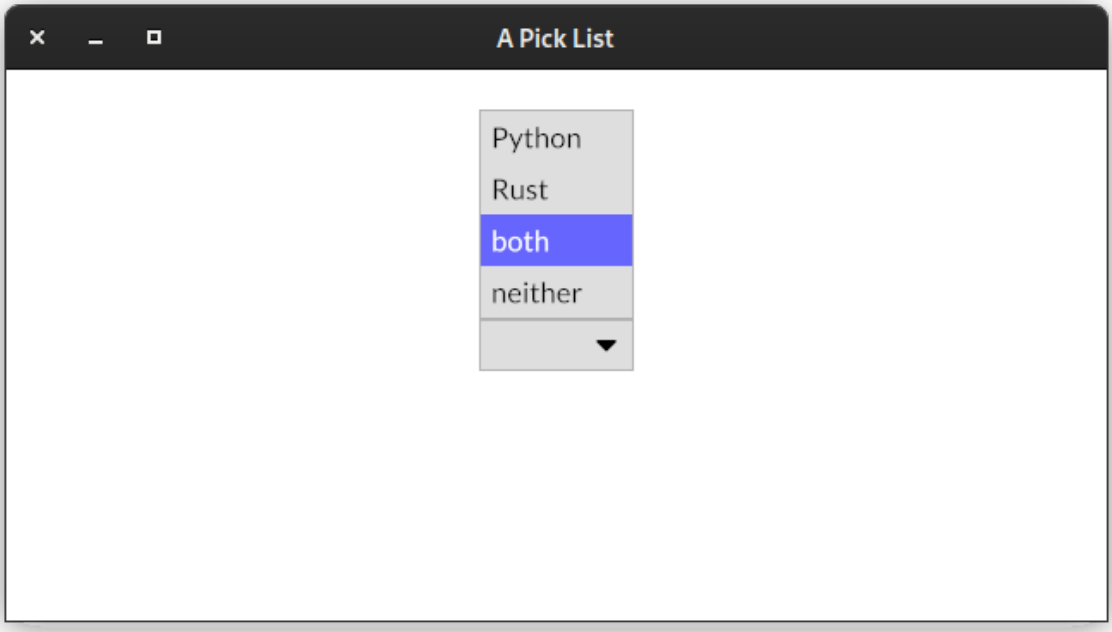
Parameters

- **token** (*object*) – When the user select a value, a message `(token, new_value)` is sent to the app's *update()* method.
- **state** (PickListState) – Current state of the pick list. The same object must be given between calls.
- **selected** (*Optional[str]*) – The currently selected value.
- **options** (*Iterable[Optional[str]]*) – Values to select from.
- **text_size** (*Optional[int]*) – The text size of the pick list.
- **font** (*Optional[Font]*) – Font of the pick list.
- **style** (*Optional[PickListStyle]*) – Style of the pick list.

Returns The newly created pick list.

Return type *Element*

Example




```
from asyncio import sleep

from pyiced import (
    Align, container, IcedApp, Length, pick_list, PickListState, text,
)


class PickListExample(IcedApp):
    class settings:
        class window:
```

(continues on next page)

(continued from previous page)

```
            size = (640, 320)

    def __init__(self):
        self.__pick_list_state = PickListState()
        self.__selected = None
        self.__enabled = True

    def title(self):
        return 'A Pick List'

    def view(self):
        if self.__enabled:
            element = pick_list(
                'select',
                self.__pick_list_state,
                self.__selected,
                ['Python', 'Rust', 'both', 'neither'],
            )
        else:
            element = text(':-(')

        return container(
            element,
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    def update(self, msg, clipboard):
        match msg:
            case 'select', 'neither':
                self.__enabled = False
                return [reenable()]
            case 'select', value:
                self.__selected = value
            case 'enable':
                self.__enabled = True


async def reenable():
    await sleep(2.0)
    return 'enable'


if __name__ == '__main__':
    PickListExample().run()
```

See also:

iced_native::widget::pick_list::PickList

pyiced.**progress_bar**(*start*, *end*, *value*, ***, *width=None*, *height=None*, *style=None*)

A bar that displays progress.

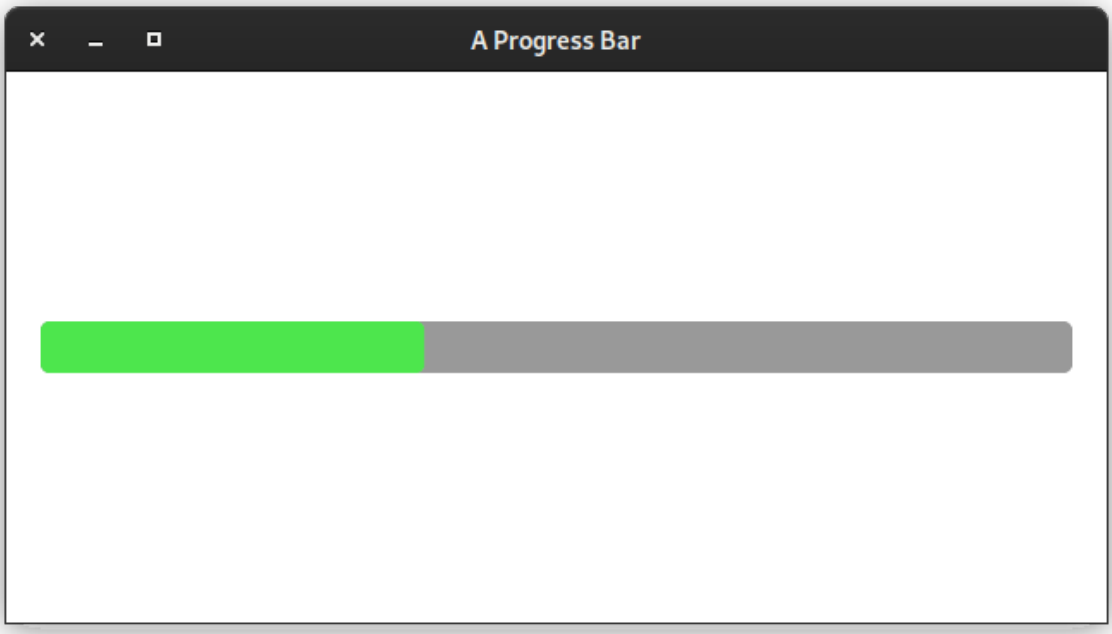
Parameters

- **start** (*float*) – Minimum value inside the value range.
- **end** (*float*) – Maximum value inside the value range.
- **value** (*float*) – Current value of the progress bar.
- **width** (*Optional[Length]*) – Width of the progress bar.
- **height** (*Optional[Length]*) – Height of the progress bar.
- **style** (*Optional[ProgressBarStyleSheet]*) – Style of the progress bar.

Returns The newly created progress bar.

Return type *Element*

Example




```
from datetime import timedelta

from pyiced import Align, container, every, IcedApp, Length, progress_bar


class ProgressBarExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__value = 0.0

    def title(self):
        return 'A Progress Bar'
```

(continues on next page)

(continued from previous page)

```
    def subscriptions(self):
        if self.__value < 1:
            return [every(timedelta(milliseconds=10), 'progress')]

    def update(self, msg, clipboard):
        match msg:
            case ('progress', _):
                self.__value = (self.__value + 0.001)

    def view(self):
        return container(
            progress_bar(0, 1, self.__value),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


if __name__ == '__main__':
    ProgressBarExample().run()
```

See also:

iced_native::widget::progress_bar::ProgressBar

pyiced.**radio**(*token*, *selected*, *value*, *label*, ***, *size=None*, *width=None*, *spacing=None*, *text_size=None*, *style=None*)

A circular button representing a choice.

Parameters

- **token** (*object*) – When the user select this choice, a message `(token, value)` is sent to the app's *update()* method.
- **selected** (*Optional[int]*) – The identifier of the currently selected option.
- **value** (*int*) – Identifier of the option.
- **label** (*str*) – Label next to the radio button.
- **size** (*Optional[int]*) – The diameter of the circle.
- **width** (*Optional[Length]*) – The width including the text.
- **spacing** (*Optional[int]*) – The spacing between the radio button and its text.
- **text_size** (*Optional[int]*) – The size of the text.
- **style** (*Optional[RadioStyleSheet]*) – Style of the radio button.

Returns The newly created radio button.

Return type *Element*

Example


```
from pyiced import column, css_color, IcedApp, Length, radio, text


class RadioExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__season = None

    def title(self):
        return 'Radio Example'

    def background_color(self):
        match self.__season:
            case 1:
                return css_color.MEDIUMSPRINGGREEN
            case 2:
                return css_color.LIGHTGOLDENRODYELLOW
            case 3:
                return css_color.GOLDENROD
            case 4:
                return css_color.GHOSTWHITE

    def update(self, msg, clipboard):
        match msg:
            case 'select', value:
```

(continues on next page)

(continued from previous page)

```
                self.__season = value

    def view(self):
        return column(
            [
                text("What's your favorite season?"),
                radio('select', self.__season, 1, 'Spring'),
                radio('select', self.__season, 2, 'Summer'),
                radio('select', self.__season, 3, 'Fall'),
                radio('select', self.__season, 4, 'Winter'),
            ],
            padding=20, spacing=5,
            width=Length.FILL, height=Length.FILL,
        )


if __name__ == '__main__':
    RadioExample().run()
```

See also:

iced_native::widget::radio::Radio

pyiced.**row**(*children*, *, *spacing=None*, *padding=None*, *width=None*, *height=None*, *max_width=None*, *max_height=None*, *align_items=None*)

A container that distributes its contents horizontally.

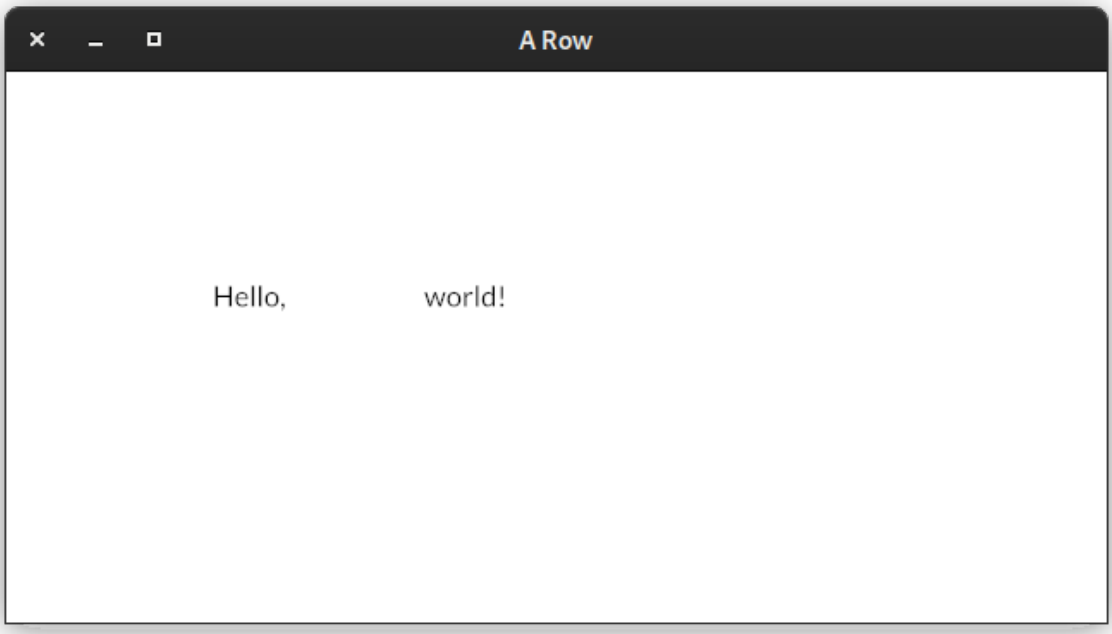
Parameters

- **children** (*Iterable[Optional[Element]]*) – Create the row with the given elements.
- **spacing** (*Optional[int]*) – Sets the horizontal spacing between elements.
- **padding** (*Optional[int]*) – Padding of the row.
- **width** (*Optional[Length]*) – Width of the row.
- **height** (*Optional[Length]*) – Height of the row.
- **max_width** (*Optional[int]*) – Maximum width of the row.
- **max_height** (*Optional[int]*) – Maximum height of the row.
- **align_items** (*Optional[Align]*) – Vertical alignment of the contents of the row.

Returns The newly created row.

Return type *Element*

Example




```
from pyiced import IcedApp, row, text


class RowExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def title(self):
        return 'A Row'

    def view(self):
        return row(
            [text('Hello,'), text('world!')],
            padding=120, spacing=80,
        )


if __name__ == '__main__':
    RowExample().run()
```

See also:

iced_native::widget::row::Row

pyiced.**rule**(*, *horizontal=None*, *vertical=None*, *style=None*)

Display a horizontal or vertical rule for dividing content.

Parameters

- **horizontal** (*Optional[int]*) – Creates a horizontal rule for dividing content by the given

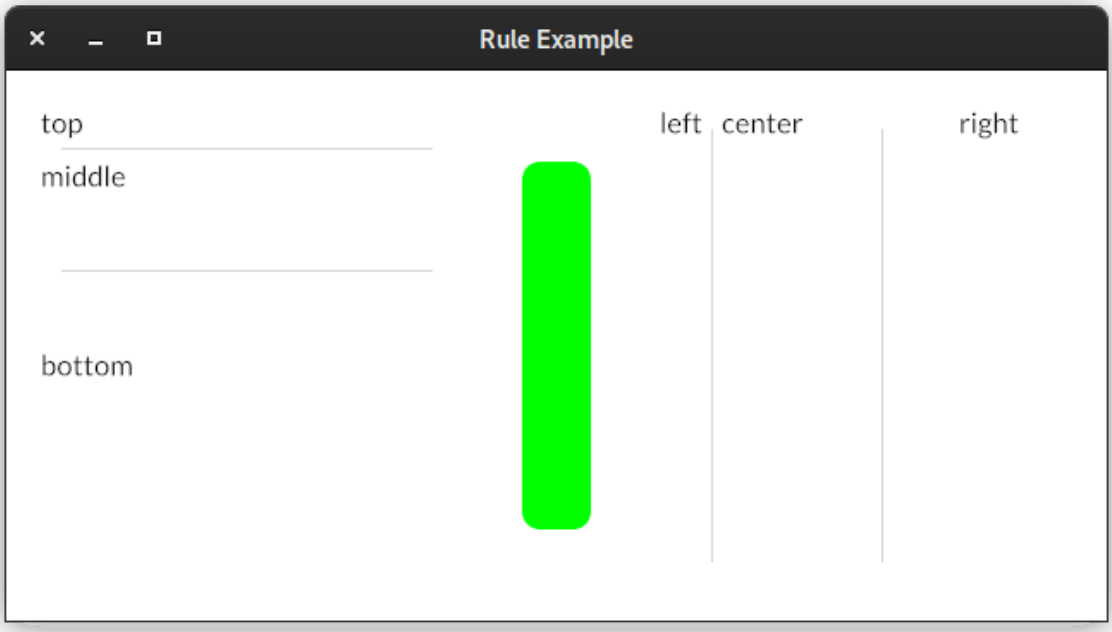
vertical spacing.

- **vertical** (*Optional[int]*) – Creates a vertical rule for dividing content by the given horizontal spacing.
- **style** (*Optional[RuleStyleSheet]*) – The style of the rule.

Returns The newly created divider.

Return type *Element*

Example




```
from pyiced import (
    Color, column, every, FillMode, IcedApp, Length, row, rule,
    RuleStyleSheet, text,
)


class RuleExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def new(self):
        self.__percent = 0

    def title(self):
        return 'Rule Example'

    def subscriptions(self):
```

(continues on next page)

(continued from previous page)

```
        return [every(0.010, 'tick')]

    def view(self):
        vertical = column(
            [
                text('top'),
                rule(horizontal=1),
                text('middle'),
                rule(horizontal=80),
                text('bottom'),
            ],
            padding=20, spacing=5,
            width=Length.FILL, height=Length.FILL,
        )
        separator = rule(
            vertical=80,
            style=RuleStyleSheet(
                color=Color(0, 1, 0),
                width=40,
                radius=10,
                fill_mode=FillMode.percent(self.__percent),
            ),
        )
        horizontal = row(
            [
                text('left'),
                rule(vertical=1),
                text('center'),
                rule(vertical=80),
                text('right'),
            ],
            padding=20, spacing=5,
            width=Length.FILL, height=Length.FILL,
        )
        return row([vertical, separator, horizontal])

    def update(self, msg, clipboard):
        match msg:
            case ('tick', _):
                self.__percent = (self.__percent + 1) % 100


if __name__ == '__main__':
    RuleExample().run()
```

See also:

iced_native::widget::rule::Rule

pyiced.**scrollable**(*state*, *children*, ***, *spacing=None*, *padding=None*, *width=None*, *height=None*, *max_width=None*, *max_heigth=None*, *align_items=None*, *scrollbar_width=None*, *scrollbar_margin=None*, *scroller_width=None*, *style=None*)

A widget that can vertically display an infinite amount of content with a scrollbar.

Parameters

- **state** (ScrollableState) – Current state of the scroll container. The same object must be given between calls.
- **children** (*Iterable[Optional[Element]]*) – Elements of the scrollable *column()*.
- **spacing** (*Optional[int]*) – Vertical spacing between elements.
- **padding** (*Optional[int]*) – Padding of the Scrollable.
- **width** (*Optional[Length]*) – Width of the scrollable.
- **height** (*Optional[Length]*) – Height of the scrollable.
- **max_width** (*Optional[int]*) – Maximum width of the scrollable.
- **max_height** (*Optional[int]*) – Maximum height of the scrollable in pixels.
- **align_items** (*Optional[Align]*) – Horizontal alignment of the contents of the scrollable.
- **scrollbar_width** (*Optional[int]*) – Scrollbar width of the Scrollable. Silently enforces a minimum value of 1.
- **scrollbar_margin** (*Optional[int]*) – Scrollbar margin of the scrollable.
- **scroller_width** (*Optional[int]*) – Scroller width of the scrollable. Silently enforces a minimum value of 1.
- **style** (*Optional[ScrollableStyleSheet]*) – The style of the scrollable.

Returns The newly created scrollable widget.

Return type *Element*

See also:

iced_native::widget::scrollable::Scrollable

pyiced.**slider**(*token*, *state*, *start*, *end*, *value*, *step=1.0*, ***, *width=None*, *height=None*, *style=None*)

An horizontal bar and a handle that selects a single value from a range of values.

Parameters

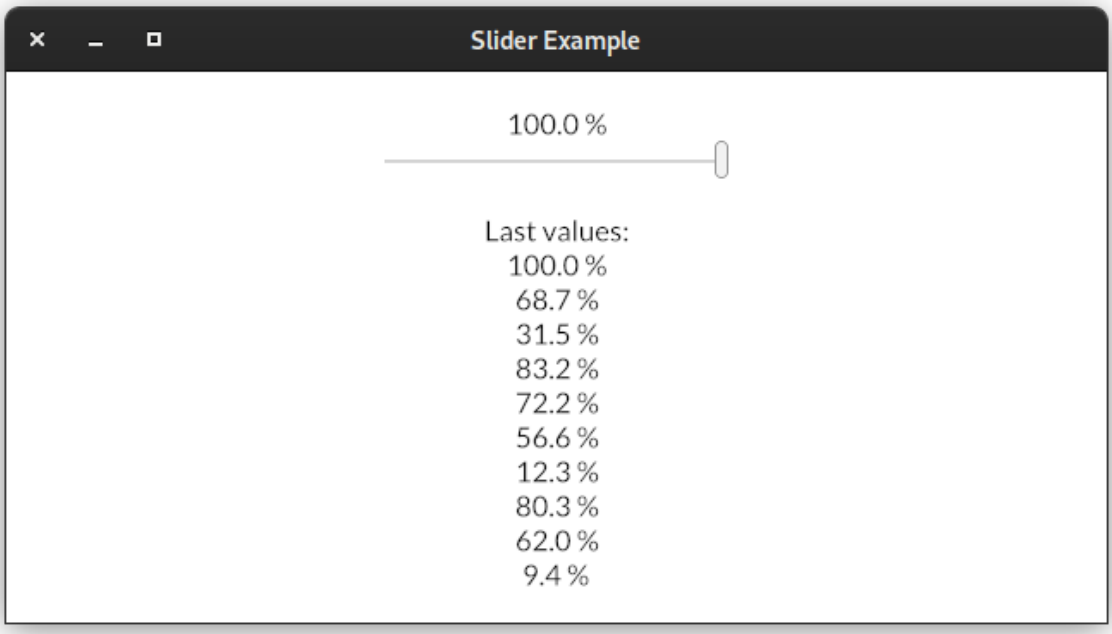
- **token** (*object*) – When the user select a value, a message `(token, new_value)` is sent to the app's *update()* method.

  When the user releases the pressed slider `(token, None, 'release')` is sent.
- **state** (SliderState) – Current state of the slider. The same object must be given between calls.
- **start** (*float*) – Smallest value inside the range.
- **end** (*float*) – Biggest value inside the range.
- **value** (*float*) – Current value.
- **step** (*float*) – Step size of the slider.
- **width** (*Optional[Length]*) – Width of the slider.
- **height** (*Optional[int]*) – Height of the slider.
- **style** (SliderStyleSheet) – The normal style of the slider.

Returns The newly created slider.

Return type *Element*

Example




```
from pyiced import (
    Align, column, container, IcedApp, Length, SliderState, slider, text,
)


class SliderApp(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__state = SliderState()
        self.__value = 0.5
        self.__messages = [' '] * 10

    def title(self):
        return 'Slider Example'

    def view(self):
        return container(
            column(
                [
                    text(f'{self.__value * 100:.1f} %'),
                    slider(
                        'slider', self.__state, 0, 1, self.__value, 0.0001,
                        width=Length.units(200),
                    ),
                    text(' '),
```

(continues on next page)

(continued from previous page)

```
                    text('Last values:'),
                    *map(text, self.__messages),
                ],
                align_items=Align.CENTER,
            ),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )

    def update(self, msg, clipboard):
        match msg:
            case 'slider', value:
                self.__value = value
            case 'slider', None, 'release':
                self.__messages.pop()
                self.__messages[:0] = (f'{self.__value * 100:.1f} %',)


if __name__ == '__main__':
    SliderApp().run()
```

See also:

iced_native::widget::slider::Slider

pyiced.**space**(*, *width=None*, *height=None*)

An amount of empty space.

It can be useful if you want to fill some space with nothing.

Parameters

- **width** (*Optional[Length]*) – Creates an amount of horizontal space.
- **height** (*Optional[Length]*) – Creates an amount of vertical space.

Returns The newly created empty space.

Return type *Element*

See also:

iced_native::widget::space::Space

pyiced.**svg**(*handle*, *, *width=None*, *height=None*)

A vector graphics image.

An SVG image resizes smoothly without losing any quality.

SVG images can have a considerable rendering cost when resized, specially when they are complex.

Parameters

- **handle** (SvgHandle) – The handle of the image.
- **width** (*Optional[Length]*) – The width of the image.
- **heigth** (*Optional[Length]*) – The height of the image.

Returns The newly created SVG image.

Return type *Element*

Example


```
from asyncio import open_connection
from contextlib import closing

from pyiced import Align, container, IcedApp, Length, svg, SvgHandle, text


class SvgExample(IcedApp):
    def __init__(self):
        self.__handle = None

    class settings:
        class window:
            size = (640, 320)

    def title(self):
        return 'An SVG'

    def new(self):
        return [load_svg()]

    def update(self, msg, clipboard):
        match msg:
            case ('SvgHandle', handle):
                self.__handle = handle

    def view(self):
        if self.__handle is None:
            return text('Loading ...')
```

(continues on next page)

(continued from previous page)

```
        return container(
            svg(
                self.__handle,
                height=Length.units(300), width=Length.units(300),
            ),
            align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


async def load_svg():
    HOST = 'raw.githubusercontent.com'
    PATH = '/iced-rs/iced/master/docs/logo.svg'

    query = (
        f"GET {PATH} HTTP/1.0\r\n"
        f"Host: {HOST}\r\n"
        f"Connection: closed\r\n"
        f"User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)\r\n"
        f"\r\n"
    ).encode('US-ASCII')

    reader, writer = await open_connection(HOST, 443, ssl=True)
    with closing(writer):
        writer.write(query)
        await writer.drain()
        while (await reader.readline()) != b'\r\n':
            continue

        data = await reader.read()
    await writer.wait_closed()

    return ('SvgHandle', SvgHandle.from_memory(data))


if __name__ == '__main__':
    SvgExample().run()
```

See also:

iced_native::widget::svg::Svg

pyiced.**text**(*label*, *, *size=None*, *color=None*, *font=None*, *width=None*, *height=None*, *horizontal_alignment=None*, *vertical_alignment=None*)

A paragraph of text.

Parameters

- **label** (*str*) – The text to display.
- **size** (*Optional[int]*) – The size of the text.
- **color** (*Optional[Color]*) – The color of the text.
- **font** (*Optional[Font]*) – The Font of the text.

- **width** (*Optional[Length]*) – The width of the text boundaries
- **height** (*Optional[Length]*) – The height of the text boundaries
- **horizontal_alignment** (*Optional[HorizontalAlignment]*) – The horizontal alignment of the text.
- **vertical_alignment** (*Optional[VerticalAlignment]*) – The vertical alignment of the Text

Returns The newly created text label.

Return type *Element*

See also:

iced_native::widget::text::Text

pyiced.**text_input**(*token*, *state*, *placeholder*, *value*, *, *font=None*, *width=None*, *max_width=None*, *padding=None*, *size=None*, *password=False*, *style=None*)

A field that can be filled with text.

Parameters

- **token** (*object*) – When the user changes the text, a message `(token, new_value)` is sent to the app's *update()* method.

  When the user hits enter, a message `(token, None, 'submit')` is sent.
- **state** (TextInputState) – Current state of the input element. The same object must be given between calls.
- **placeholder** (*str*) – Placeholder text for an element input.
- **value** (*str*) – Current value of the input element.
- **font** (*Optional[Font]*) – The font of the text.
- **width** (*Optional[Length]*) – The width of the input element.
- **max_width** (*Optional[int]*) – The maximum width of the input element.
- **padding** (*Optional[int]*) – The padding of the input element.
- **size** (*Optional[int]*) – The text size of the input element.
- **password** (*bool*) – If set to True, the input element becomes a secure password input.
- **style** (*Optional[TextInputStyleSheet]*) – Style of the text input.

Returns The newly created text input element.

Return type *Element*

See also:

iced_native::widget::text_input::TextInput

pyiced.**tooltip**(*content*, *tooltip*, *position*, *, *font=None*, *size=None*, *gap=None*, *padding=None*, *style=None*)

Make a tooltip.

Parameters

- **content** (Element) – Contained element that has a tooltip.
- **tooltip** (*str*) – Tooltip text to display.
- **position** (TooltipPosition) – The position of the tooltip.

- **font** (*Optional[Font]*) – The font of the tooltip.
- **size** (*Optional[int]*) – The size of the text of the tooltip.
- **gap** (*Optional[int]*) – The gap between the content and its tooltip.
- **padding** (*Optional[int]*) – TODO
- **style** (*Optional[ContainerStyleSheet]*) – The style of the tooltip.

Returns The newly created tooltip.

Return type *Element*

See also:

iced_native::widget::tooltip::Tooltip

4.4 State Objects

To keep the state of an *Element* across multiple invocations of *view()*, e.g. the cursor position in a *text_input()*, you have to supply a state object.

Warning: If the same state object is used for multiple elements in the same *view()* call, only the first element get displayed. All and further elements with the same state become *no_element()*.

4.4.1 Overview

ButtonState()	The state of a *button()*.
PickListState()	The state of a *pick_list()*.
ScrollableState()	The state of a *scrollable()*.
SliderState()	The state of a *slider()*.
TextInputState()	The state of a *text_input()*.

4.4.2 Details

class pyiced.**ButtonState**
The state of a *button()*.

class pyiced.**PickListState**
The state of a *pick_list()*.

class pyiced.**ScrollableState**
The state of a *scrollable()*.

Warning: If the state is currently in use, calling its methods will fail.

is_scroll_box_touched()
Returns whether the scroll box is currently touched or not.

Returns Yes or no

Return type bool

`is_scroller_grabbed()`
Returns whether the scroller is currently grabbed or not.

Returns Yes or no

Return type bool

`offset(bounds, content_bounds)`
The current scrolling offset of the ScrollableState, given the bounds of the Scrollable and its contents.

Parameters

- **bounds** (Rectangle) – TODO
- **content_bounds** (Rectangle) – TODO

Returns The scrolling offset.

Return type int

`scroll(delta_y, bounds, content_bounds)`
Apply a scrolling offset to the current ScrollableState, given the bounds of the Scrollable and its contents.

Parameters

- **delta_y** (*float*) – TODO
- **bounds** (Rectangle) – TODO
- **content_bounds** (Rectangle) – TODO

`scroll_to(percentage, bounds, content_bounds)`
Moves the scroll position to a relative amount, given the bounds of the Scrollable and its contents.

0.0 represents scrollbar at the top, while 1.0 represents scrollbar at the bottom.

Parameters

- **percentage** (*float*) – TODO
- **bounds** (Rectangle) – TODO
- **content_bounds** (Rectangle) – TODO

`class pyiced.SliderState`
The state of a *slider()*.

`class pyiced.TextInputState`
The state of a *text_input()*.

See also:

iced_native::widget::text_input::State

`focus()`
Focuses the *text_input()*.

> **Warning:** If the state is currently in use, the method will fail.

`is_focused()`
Returns whether the *text_input()* is currently focused or not.

> **Warning:** If the state is currently in use, the method will fail.

Returns Yes or no

Return type bool

`move_cursor_to`(*position*)
Moves the *TextInputCursor()* of the `TextInput` to an arbitrary location.

The result is measured in terms of graphems, not bytes or codepoints!

See also:

pyiced.TextInputState.state()

> **Warning:** If the state is currently in use, the method will fail.

Parameters **position** (*int*) – The new cursor position.

`move_cursor_to_end`()
Moves the *TextInputCursor()* of the `TextInput` to the end of the input text.

> **Warning:** If the state is currently in use, the method will fail.

`move_cursor_to_front`()
Moves the *TextInputCursor()* of the `TextInput` to the front of the input text.

> **Warning:** If the state is currently in use, the method will fail.

`selection`(*value*)
Get the selected text.

> **Warning:** If the state is currently in use, the method will fail.

Parameters **value** (*str*) – The current value of the *text_input()*.

Returns The selected text. May be empty.

Return type str

`state`(*value*)
Get the state of the *TextInputCursor()*.

The result is measured in terms of graphems, not bytes or codepoints!

> **Warning:** If the state is currently in use, the method will fail.

See also:

pyiced.TextInputState.move_cursor_to()

Returns

- *int* – The current cursor position when there's no selection.
- *Tuple[int, int]* – The selected text range.

unfocus()
Unfocuses the *text_input()*.

Warning: If the state is currently in use, the method will fail.

4.5 Values and Enums

4.5.1 Overview

Align	Alignment on an axis of a container.
Clipboard	A buffer for short-term storage and transfer within and between applications.
FillMode	The fill mode of a rule.
HorizontalAlignment	The horizontal alignment of some resource.
ImageHandle	An *pyiced.image()* handle.
Instant()	A measurement of a monotonically nondecreasing clock.
Length	The strategy used to fill space in a specific dimension.
Line(color, width)	A line.
Point(x, y)	A 2D point.
Rectangle(top_left, size)	A rectangle.
Size(width, height)	An amount of space in 2 dimensions.
SliderHandle(shape, color, border_width, ...)	The appearance of the handle of a slider.
SliderHandleShape	The shape of the handle of a slider.
SvgHandle	An *svg()* handle.
TextInputCursor(state)	A representation of cursor position in a *text_input()*.
TooltipPosition	The position of the tooltip.
VerticalAlignment	The vertical alignment of some resource.

4.5.2 Details

class pyiced.**Align**
Alignment on an axis of a container.

See also:

iced::Align

START
Align at the start of the axis.

CENTER
Align at the center of the axis.

END
Align at the end of the axis.

class pyiced.**Clipboard**
A buffer for short-term storage and transfer within and between applications.

> **Warning:** The clipboard is only valid during the call to *pyiced.IcedApp.update()*.

See also:

iced::Clipboard

read()
Reads the current content of the clipboard as text.

Returns The current contents of the clipboard.

Return type Optional[str]

write(*value*)
Writes the given text contents to the clipboard.

Parameters **value** (*str*) – The new contents of the clipboard.

class pyiced.**FillMode**
The fill mode of a rule.

See also:

iced::widget::rule::FillMode

FULL = FillMode.FULL

static asymmetric_padding(*first_pad*, *second_pad*)
Different offset on each end of the rule.

Parameters

- **first_pad** (*int*) – top or left, length units
- **second_pad** (*int*) – the other direction, length units

static padded(*i*)
Uniform offset from each end.

Parameters **i** (*int*) – Length units.

static percent(*percentage*)
Fill a percent of the length of the container. The rule will be centered in that container.

Parameters **percentage** (*float*) – The range is [0.0, 100.0]. The value gets clamped in this range automatically.

class pyiced.**HorizontalAlignment**
The horizontal alignment of some resource.

See also:

iced::HorizontalAlignment

LEFT
Align left

CENTER
Horizontally centered

RIGHT
Align right

class pyiced.**ImageHandle**
An *pyiced.image()* handle.

See also:

iced_native::widget::image::Handle

static from_memory(*bytes*)
Creates an image handle containing the image data directly.

Parameters **bytes** (*bytes-like*) – The data of the image file.

Returns The new image handle.

Return type *ImageHandle*

static from_path(*path*)
Creates an image handle pointing to the image of the given path.

Parameters **path** (*pathlib.Path*) – The path of the image file.

Returns The new image handle.

Return type *ImageHandle*

class pyiced.**Instant**
A measurement of a monotonically nondecreasing clock. Opaque and useful only with duration.

- You can add/substract a number of seconds as `float` to/from an instant to get a new instant.
- You can add/substract a `timedelta` to/from an instant to get a new instant.
- You can substract two instants to get the number of seconds as `float` between them: `later - earlier = seconds`.

See also:

std::time::Instant

class pyiced.**Length**
The strategy used to fill space in a specific dimension.

See also:

iced::Length

FILL = Length.FILL

SHRINK = Length.SHRINK

static fill_portion(*i*)
Fill a portion of the remaining space relative to other elements.

static units(*i*)
Fill a fixed amount of space.

class pyiced.**Line**(*color*, *width*)
A line.

It is normally used to define the highlight of something, like a split.

Parameters

- **color** (Color) – The color of the line.
- **width** (*float*) – The width of the line.

See also:

iced::widget::pane_grid::Line

color
The color of the line.

Returns The "color" parameter given when constructing this line.

Return type *Color*

width
The width of the line.

Returns The "width" parameter given when constructing this line.

Return type float

class pyiced.**Point**(*x*, *y*)
A 2D point.

Parameters

- **x** (*float*) – The X coordinate.
- **y** (*float*) – The Y coordinate.

See also:

iced::Point

ORIGIN = Point(0, 0)

distance(*to*)
Computes the distance to another point.

Parameters **to** (Point) – The other point.

x
The X coordinate.

Returns The "x" parameter given when constructing this point.

Return type float

y
The Y coordinate.

Returns The "y" parameter given when constructing this point.

Return type float

class pyiced.**Rectangle**(*top_left*, *size*)
A rectangle.

See also:

iced::Rectangle

Parameters

- **top_left** (Point) – The top-left corner.
- **size** (Size) – The size of the rectangle.

height
Height of the rectangle.

Returns The "size.height" parameter given when constructing this point.

Return type float

size
The size of the rectangle.

Returns The "size" parameter given when constructing this point.

Return type *Size*

top_left
The top-left corner.

Returns The "top_left" parameter given when constructing this point.

Return type *Point*

width
Width of the rectangle.

Returns The "size.width" parameter given when constructing this point.

Return type float

static with_size(*size*)
Creates a new Rectangle with its top-left corner at the origin and with the provided Size.

Parameters **size** (Size) – Size of the new Rectangle

Returns The new Rectangle.

Return type *Rectangle*

x
X coordinate of the top-left corner.

Returns The "top_left.x" parameter given when constructing this point.

Return type float

y
Y coordinate of the top-left corner.

Returns The "top_left.y" parameter given when constructing this point.

Return type float

class pyiced.**Size**(*width*, *height*)
An amount of space in 2 dimensions.

Parameters

- **width** (*float*) – The width.
- **height** (*float*) – The height.

See also:

iced::Size

INFINITY = Size(inf, inf)

UNIT = Size(1.0, 1.0)

ZERO = Size(0.0, 0.0)

height
The height.

Returns The "height" parameter given when constructing this size.

Return type float

pad(*padding*)
Increments the Size to account for the given padding.

Parameters **padding** (*float*) – The other size.

width
The width.

Returns The "width" parameter given when constructing this size.

Return type float

class pyiced.**SliderHandle**(*shape*, *color*, *border_width*, *border_color*)
The appearance of the handle of a slider.

Parameters

- **shape** (SliderHandleShape) – The color of the slider_handle.
- **color** (Color) – The width of the slider_handle.
- **border_width** (*float*) – The width of the slider_handle.
- **border_color** (Color) – The width of the slider_handle.

See also:

iced::widget::slider::Handle

class pyiced.**SliderHandleShape**
The shape of the handle of a slider.

See also:

iced::widget::slider::HandleShape

static circle(*radius*)
A circle.

Parameters **radius** (*float*) – The radius of the circle

Returns A slider handle in the shape of a circle.

Return type *SliderHandleShape*

static rectangle(*width*, *border_radius*)
A rectangle.

Parameters

- **width** (*float*) – The length of an edge.
- **border_radius** (*float*) – The border radius.

Returns A slider handle in the shape of a rectangle.

Return type *SliderHandleShape*

class pyiced.**SvgHandle**
An *svg()* handle.

See also:

iced::widget::svg::Handle

static from_memory(*bytes*)
Creates an SVG handle containing the image data directly.

Parameters **bytes** (*bytes-like*) – Creates an SVG Handle from raw bytes containing either an SVG string or gzip compressed data.

This is useful if you already have your SVG data in-memory, maybe because you downloaded or generated it procedurally.

Returns An SVG handle usable in *svg()*.

Return type *SvgHandle*

static from_path(*path*)
Creates an SVG Handle pointing to the vector image of the given path.

Parameters **path** (*path-like*) – Creates an SVG Handle pointing to the vector image of the given path.

Returns An SVG handle usable in *svg()*.

Return type *SvgHandle*

class pyiced.**TextInputCursor**(*state*)
A representation of cursor position in a *text_input()*.

There should be no reason to create or inspect this object directly.

Parameters **state** (TextInputState) – Text input state to inspect.

See also:

iced_native::widget::text_input::cursor::Cursor

selection(*value*)
Get the selected text.

> **Warning:** If the state is currently in use, the method will fail.

Parameters **value** (*str*) – The current value of the *text_input()*.

Returns The selected text. May be empty.

Return type str

state(*value*)
Get the state of the *TextInputCursor()*.

The result is measured in terms of graphems, not bytes or codepoints!

> **Warning:** If the state is currently in use, the method will fail.

See also:

pyiced.TextInputState.move_cursor_to()

Returns

- *int* – The current cursor position when there's no selection.
- *Tuple[int, int]* – The selected text range.

class pyiced.**TooltipPosition**

The position of the tooltip.

See also:

iced::widget::tooltip::Position

FOLLOW_CURSOR

The tooltip will follow the cursor.

TOP

The tooltip will appear on the top of the widget.

BOTTOM

The tooltip will appear on the bottom of the widget.

LEFT

The tooltip will appear on the left of the widget.

RIGHT

The tooltip will appear on the right of the widget.

class pyiced.**VerticalAlignment**

The vertical alignment of some resource.

See also:

iced::VerticalAlignment

TOP

Align top

CENTER

Vertically centered

BOTTOM

Align bottom

4.6 Colors

4.6.1 Overview

Color(r, g, b[, a])	A color in the sRGB color space.

4.6.2 Details

class pyiced.**Color**(*r*, *g*, *b*, *a=1.0*)

A color in the sRGB color space.

Parameters

- **r** (*float*) – Red component, 0.0 – 1.0
- **g** (*float*) – Green component, 0.0 – 1.0
- **b** (*float*) – Blue component, 0.0 – 1.0
- **a** (*float*) – Alpha channel, 0.0 – 1.0 (0.0 = transparent; 1.0 = opaque)

BLACK = Color(0, 0, 0)

TRANSPARENT = Color(0, 0, 0, a=0)

WHITE = Color(1, 1, 1)

a

Alpha channel, 0.0 – 1.0 (0.0 = transparent; 1.0 = opaque)

Returns Color channel value

Return type float

b

Blue component, 0.0 – 1.0

Returns Color channel value

Return type float

g

Green component, 0.0 – 1.0

Returns Color channel value

Return type float

r

Red component, 0.0 – 1.0

Returns Color channel value

Return type float

4.6.3 Named Colors

pyiced.css_color exports *pyiced.Color* constants for all 148 named CSS Color Module Level 4 colors.

pyiced.css_color.**ALICEBLUE**

pyiced.css_color.**ANTIQUEWHITE**

pyiced.css_color.**AQUA**

pyiced.css_color.**AQUAMARINE**

pyiced.css_color.**AZURE**

pyiced.css_color.**BEIGE**

pyiced.css_color.**BISQUE**

pyiced.css_color.**BLACK**

pyiced.css_color.**BLANCHEDALMOND**

pyiced.css_color.**BLUE**

pyiced.css_color.**BLUEVIOLET**

pyiced.css_color.**BROWN**

pyiced.css_color.**BURLYWOOD**

pyiced.css_color.**CADETBLUE**

pyiced.css_color.**CHARTREUSE**

pyiced.css_color.**CHOCOLATE**

pyiced.css_color.**CORAL**

pyiced.css_color.**CORNFLOWERBLUE**

pyiced.css_color.**CORNSILK**

pyiced.css_color.**CRIMSON**

pyiced.css_color.**CYAN**

pyiced.css_color.**DARKBLUE**

pyiced.css_color.**DARKCYAN**

pyiced.css_color.**DARKGOLDENROD**

pyiced.css_color.**DARKGRAY**

pyiced.css_color.**DARKGREEN**

pyiced.css_color.**DARKGREY**

pyiced.css_color.**DARKKHAKI**

pyiced.css_color.**DARKMAGENTA**

pyiced.css_color.**DARKOLIVEGREEN**

pyiced.css_color.**DARKORANGE**

pyiced.css_color.**DARKORCHID**

pyiced.css_color.**DARKRED**

pyiced.css_color.**DARKSALMON**

pyiced.css_color.**DARKSEAGREEN**

pyiced.css_color.**DARKSLATEBLUE**

pyiced.css_color.**DARKSLATEGRAY**

pyiced.css_color.**DARKSLATEGREY**

pyiced.css_color.**DARKTURQUOISE**

pyiced.css_color.**DARKVIOLET**

pyiced.css_color.**DEEPPINK**

pyiced.css_color.**DEEPSKYBLUE**

pyiced.css_color.**DIMGRAY**

pyiced.css_color.**DIMGREY**

pyiced.css_color.**DODGERBLUE**

pyiced.css_color.**FIREBRICK**

pyiced.css_color.**FLORALWHITE**

pyiced.css_color.**FORESTGREEN**

pyiced.css_color.**FUCHSIA**

pyiced.css_color.**GAINSBORO**

pyiced.css_color.**GHOSTWHITE**

pyiced.css_color.**GOLD**

pyiced.css_color.**GOLDENROD**

pyiced.css_color.**GRAY**

pyiced.css_color.**GREEN**

pyiced.css_color.**GREENYELLOW**

pyiced.css_color.**GREY**

pyiced.css_color.**HONEYDEW**

pyiced.css_color.**HOTPINK**

pyiced.css_color.**INDIANRED**

pyiced.css_color.**INDIGO**

pyiced.css_color.**IVORY**

pyiced.css_color.**KHAKI**

pyiced.css_color.**LAVENDER**

pyiced.css_color.**LAVENDERBLUSH**

pyiced.css_color.**LAWNGREEN**

pyiced.css_color.**LEMONCHIFFON**

pyiced.css_color.**LIGHTBLUE**

pyiced.css_color.**LIGHTCORAL**

pyiced.css_color.**LIGHTCYAN**

pyiced.css_color.**LIGHTGOLDENRODYELLOW**

pyiced.css_color.**LIGHTGRAY**

pyiced.css_color.**LIGHTGREEN**

pyiced.css_color.**LIGHTGREY**

pyiced.css_color.**LIGHTPINK**

pyiced.css_color.**LIGHTSALMON**

pyiced.css_color.**LIGHTSEAGREEN**

pyiced.css_color.**LIGHTSKYBLUE**

pyiced.css_color.**LIGHTSLATEGRAY**

pyiced.css_color.**LIGHTSLATEGREY**

pyiced.css_color.**LIGHTSTEELBLUE**

pyiced.css_color.**LIGHTYELLOW**

pyiced.css_color.**LIME**

pyiced.css_color.**LIMEGREEN**

pyiced.css_color.**LINEN**

pyiced.css_color.**MAGENTA**

pyiced.css_color.**MAROON**

pyiced.css_color.**MEDIUMAQUAMARINE**

pyiced.css_color.**MEDIUMBLUE**

pyiced.css_color.**MEDIUMORCHID**

pyiced.css_color.**MEDIUMPURPLE**

pyiced.css_color.**MEDIUMSEAGREEN**

pyiced.css_color.**MEDIUMSLATEBLUE**

pyiced.css_color.**MEDIUMSPRINGGREEN**

pyiced.css_color.**MEDIUMTURQUOISE**

pyiced.css_color.**MEDIUMVIOLETRED**

pyiced.css_color.**MIDNIGHTBLUE**

pyiced.css_color.**MINTCREAM**

pyiced.css_color.**MISTYROSE**

pyiced.css_color.**MOCCASIN**

pyiced.css_color.**NAVAJOWHITE**

pyiced.css_color.**NAVY**

pyiced.css_color.**OLDLACE**

pyiced.css_color.**OLIVE**

pyiced.css_color.**OLIVEDRAB**

pyiced.css_color.**ORANGE**

pyiced.css_color.**ORANGERED**

pyiced.css_color.**ORCHID**

pyiced.css_color.**PALEGOLDENROD**

pyiced.css_color.**PALEGREEN**

pyiced.css_color.**PALETURQUOISE**

pyiced.css_color.**PALEVIOLETRED**

pyiced.css_color.**PAPAYAWHIP**

pyiced.css_color.**PEACHPUFF**

pyiced.css_color.**PERU**

pyiced.css_color.**PINK**

pyiced.css_color.**PLUM**

pyiced.css_color.**POWDERBLUE**

pyiced.css_color.**PURPLE**

pyiced.css_color.**REBECCAPURPLE**

pyiced.css_color.**RED**

pyiced.css_color.**ROSYBROWN**

pyiced.css_color.**ROYALBLUE**

pyiced.css_color.**SADDLEBROWN**

pyiced.css_color.**SALMON**

pyiced.css_color.**SANDYBROWN**

pyiced.css_color.**SEAGREEN**

pyiced.css_color.**SEASHELL**

pyiced.css_color.**SIENNA**

pyiced.css_color.**SILVER**

pyiced.css_color.**SKYBLUE**

pyiced.css_color.**SLATEBLUE**

pyiced.css_color.**SLATEGRAY**

pyiced.css_color.**SLATEGREY**

pyiced.css_color.**SNOW**

pyiced.css_color.**SPRINGGREEN**

pyiced.css_color.**STEELBLUE**

pyiced.css_color.**TAN**

pyiced.css_color.**TEAL**

pyiced.css_color.**THISTLE**

pyiced.css_color.**TOMATO**

pyiced.css_color.**TURQUOISE**

pyiced.css_color.**VIOLET**

pyiced.css_color.**WHEAT**

pyiced.css_color.**WHITE**

pyiced.css_color.**WHITESMOKE**

pyiced.css_color.**YELLOW**

pyiced.css_color.**YELLOWGREEN**

4.7 Fonts

4.7.1 Overview

Font(name, data)	A font.
FontFamily	A font family.
FontId	A unique per database face ID.
FontStretch	A CSS font-stretch.
FontStyle	Allows italic or oblique faces to be selected.
FontWeight	Specifies the weight of glyphs in the font, their degree of blackness or stroke thickness.
findfont([family, weight, stretch, style])	Performs a CSS-like query and returns the best matched font face.
systemfonts()	List loaded system fonts.

4.7.2 Details

class pyiced.**Font**(*name*, *data*)

A font.

The font does not get loaded multiple times, but instead the name is used to tell fonts apart. So you should use the same name for the same data in subsequent Font instance creations.

Parameters

- **name** (*str*) – The name of the external font
- **data** (*bytes-like*) – The bytes of the external font

See also:

iced::Font

> **Warning:** The font data gets interned! Even of the module is unloaded / reloaded, some memory is lost until the interpreter is restarted.

DEFAULT = Font.DEFAULT

data

Bytes data of the font

Returns

- *memoryview* – The bytes data of the font.
- *None* – For *DEFAULT*.

name

Name of the font

Returns

- *str* – The name of the font.
- *None* – For *DEFAULT*.

class pyiced.**FontFamily**
A font family.

See also:

fontdb::Family

CURSIVE = FontFamily.CURSIVE

FANTASY = FontFamily.FANTASY

MONOSPACE = FontFamily.MONOSPACE

SANSSERIF = FontFamily.SANSSERIF

SERIF = FontFamily.SERIF

class pyiced.**FontId**
A unique per database face ID.

See also:

fontdb::ID

family
Corresponds to a Font Family in a TrueType font.

load()
Loads the referenced font into memory.

Returns The Font object to be used in e.g. *view()*.

Return type *Font*

monospaced
Indicates that the font face is monospaced.

name
Corresponds to a PostScript name in a TrueType font.

stretch
A font face stretch.

style
A font face style.

weight
A font face weight.

class pyiced.**FontStretch**
A CSS font-stretch.

See also:

fontdb::Stretch

ULTRACONDENSED
50% width

EXTRACONDENSED
62.5% width

CONDENSED
75% width

SEMICONDENSED
87.5% width

NORMAL
100% width

SEMIEXPANDED
112.5% width

EXPANDED
125% width

EXTRAEXPANDED
150% width

ULTRAEXPANDED
200% width

class pyiced.**FontStyle**
Allows italic or oblique faces to be selected.

See also:

fontdb::Style

NORMAL
A face that is neither italic not obliqued.

ITALIC
A form that is generally cursive in nature.

OBLIQUE
A typically-sloped version of the regular face.

class pyiced.**FontWeight**
Specifies the weight of glyphs in the font, their degree of blackness or stroke thickness.

See also:

fontdb::Weight

BLACK = FontWeight.BLACK

BOLD = FontWeight.BOLD

EXTRABOLD = FontWeight.EXTRABOLD

EXTRALIGHT = FontWeight.EXTRALIGHT

LIGHT = FontWeight.LIGHT

MEDIUM = FontWeight.MEDIUM

NORMAL = FontWeight.NORMAL

SEMIBOLD = FontWeight.SEMIBOLD

THIN = FontWeight.THIN

value

pyiced.**findfont**(*family=None*, *weight=None*, *stretch=None*, *style=None*)
Performs a CSS-like query and returns the best matched font face.

Arguments can be given using their constants or using their CSS value, e.g.

```
>>> from pyiced import  *
>>> findfont("serif", "extra-light", "normal", "italic")
```

(continues on next page)

(continued from previous page)

```
FontId(name="TimesNewRomanPS-ItalicMT", family="Times New Roman",
       style=Italic, weight=Weight(400), stretch=Normal)
```

Parameters

- **families** (*Union[FontFamily, str, Iterable[Union[FontFamily, str]], None]*) – A prioritized (list of) font family names or generic family name(s). Defaults to *SANSSERIF*.
- **weight** (*Union[FontWeight, int, str, None]*) – Specifies the weight of glyphs in the font, their degree of blackness or stroke thickness. Defaults to *NORMAL*.
- **stretch** (*Union[FontStretch, str, None]*) – Selects a normal, condensed, or expanded face from a font family. Defaults to *NORMAL*.
- **style** (*Union[FontStyle, str, None]*) – Allows italic or oblique faces to be selected. Defaults to *NORMAL*.

Returns The best match found, if one was found.

Return type Optional[*FontId*]

See also:

fontdb::Query

pyiced.**systemfonts**()

List loaded system fonts.

Returns An iterator over all system fonts.

Return type Iterator[*FontId*]

4.8 Element Styles

4.8.1 Overview

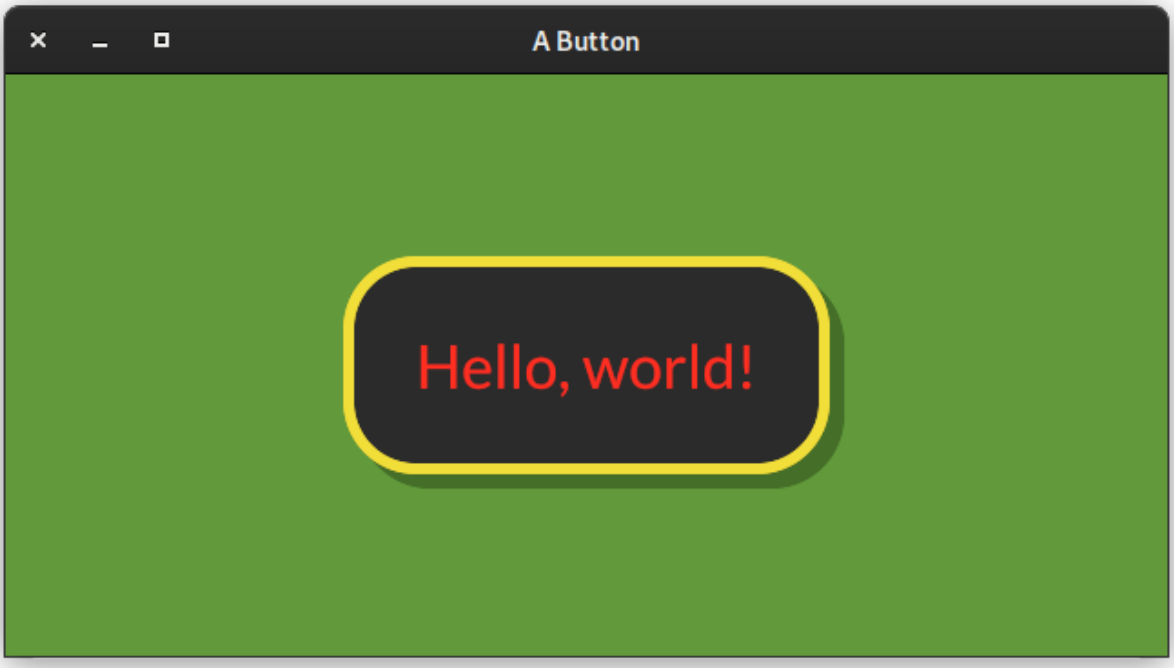
ButtonStyle([proto])	The appearance of a *button()* for a given state.
ButtonStyleSheet(active[, hovered, pressed, ...])	The appearance of a *button()*.
CheckboxStyle([proto])	The appearance of a *checkbox()* for some state.
CheckboxStyleSheet(active[, hoverered, ...])	The appearance of a *checkbox()*.
ContainerStyle	alias of *pyiced.ContainerStyleSheet*
ContainerStyleSheet([proto])	The appearance of a *container()*.
PaneGridStyle	alias of *pyiced.PaneGridStyleSheet*
PaneGridStyleSheet([proto])	The appearance of a pane_grid().
PickListMenu([proto])	The appearance of a pick list menu.
PickListStyle([proto])	The appearance of a *pick_list()* for some state.
PickListStyleSheet(menu, active[, hovered])	The appearance of a *pick_list()*.
ProgressBarStyle	alias of *pyiced.ProgressBarStyleSheet*
ProgressBarStyleSheet([proto])	The appearance of a *progress_bar()*.
RadioStyle([proto])	The appearance of a *radio()* for some state.
RadioStyleSheet(active[, hovered])	The appearance of a *radio()*.
RuleStyle	alias of *pyiced.RuleStyleSheet*

continues on next page

Table 7 – continued from previous page

RuleStyleSheet([proto])	The appearance of a *rule()*.
ScrollableStyleSheet(active[, hovered, dragging])	The appearance of a *scrollable()*.
ScrollbarStyle([proto])	The appearance a specific state of a *scrollable()*.
ScrollerStyle([proto])	The appearance of the scroller of a *scrollable()*.
SliderStyle([proto])	The appearance of a *slider()* for some state.
SliderStyleSheet(active[, hovered, dragging])	The appearance of a *slider()*.
TextInputStyle([proto])	The appearance of a *text_input()* for some state.
TextInputStyleSheet(active[, focused, ...])	The appearance of a *text_input()*.
TooltipStyle	alias of *pyiced.ContainerStyleSheet*
TooltipStyleSheet	alias of *pyiced.ContainerStyleSheet*

4.8.2 Quick Example




```
from pyiced import (
    Align, button, ButtonState, ButtonStyle, ButtonStyleSheet, Color,
    container, ContainerStyle, IcedApp, Length, text,
)


class ButtonExample(IcedApp):
    class settings:
        class window:
            size = (640, 320)

    def __init__(self):
        self.__button_state = ButtonState()
```

(continues on next page)

(continued from previous page)

```
    def title(self):
        return 'A Button'

    def view(self):
        styled_button = button(
            self.__button_state,
            text('Hello, world!', size=40),
            '',
            style=ButtonStyleSheet(ButtonStyle(
                shadow_offset=(8, 8), border_radius=40, border_width=6,
                background=Color(0.17, 0.17, 0.17),
                border_color=Color(0.95, 0.87, 0.22),
                text_color=Color(1.00, 0.18, 0.13)
            )),
            padding=40,
        )
        return container(
            styled_button,
            style=ContainerStyle(background=Color(0.38, 0.60, 0.23)),
            padding=20, align_x=Align.CENTER, align_y=Align.CENTER,
            width=Length.FILL, height=Length.FILL,
        )


if __name__ == '__main__':
    ButtonExample().run()
```

4.8.3 Details

class pyiced.**ButtonStyle**(*proto=None*, ***kwargs*)

The appearance of a *button()* for a given state.

Parameters

- **proto** (*Optional[ButtonStyleSheet]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.
- **shadow_offset** (*Tuple[float, float]*) – The button's shadow offset.
- **background** (*Optional[Color]*) – The button's background color.
- **border_radius** (*float*) – The button's border radius.
- **border_width** (*float*) – The button's border width.
- **border_color** (*Color*) – The button's border color.
- **text_color** (*Color*) – The button's text color.

See also:

iced::widget::button::Style

class pyiced.**ButtonStyleSheet**(*active*, *hovered=None*, *pressed=None*, *disabled=None*)

The appearance of a *button()*.

Parameters

- **active** (ButtonStyle) – Normal style of the button.
- **hovered** (*Optional[ButtonStyle]*) – Style of the button when the cursor is hovering over it. Defaults to a style derived from "active".
- **pressed** (*Optional[ButtonStyle]*) – Style of the button while it's pressed down. Defaults to a style derived from "active".
- **disabled** (*Optional[ButtonStyle]*) – Style of the button when no "on_press" argument was given. Defaults to a style derived from "active".

See also:

iced::widget::button::StyleSheet

active
The "active" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ButtonStyle*

disabled
The "disabled" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ButtonStyle*

hovered
The "hovered" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ButtonStyle*

pressed
The "pressed" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ButtonStyle*

class pyiced.**CheckboxStyle**(*proto=None*, ***kwargs*)
The appearance of a *checkbox()* for some state.

Parameters

- **proto** (*Optional[Union[CheckboxStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active", "hovered", "active_checked" and "hovered_checked", same as the argument for `pyiced.~CheckboxStyleSheet`.

  None is the same as "active".
- **background** (Color) – The checkbox' background color.
- **checkmark_color** (Color) – The color of the checkbox.
- **border_radius** (*float*) – The checkbox' border radius.
- **border_width** (*float*) – The checkbox' border width.
- **border_color** (Color) – The checkbox' border color.

See also:

iced::widget::checkbox::Style

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

checkmark_color
The "checkmark_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

class pyiced.**CheckboxStyleSheet**(*active*, *hoverered=None*, *active_checked=None*, *hovered_checked=None*)
The appearance of a *checkbox()*.

Parameters

- **active** (CheckboxStyle) – Normal style of this checkbox.
- **hovered** (*Optional[CheckboxStyle]*) – Style when hovering over the checkbox. Defaults to the same style as "active".
- **active_checked** (*Optional[CheckboxStyle]*) – Style of this checkbox when the checkbox is checked. Defaults to the same style as "active".
- **hovered_checked** (*Optional[CheckboxStyle]*) – Style when hovering over the checked checkbox. If None or absent, it defaults to the first argument with an explicit value in "hovered", "active_checked" or "active".

See also:

iced::widget::checkbox::StyleSheet

active
The "active" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *CheckboxStyle*

active_checked
The "active_checked" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *CheckboxStyle*

hovered
The "hovered" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *CheckboxStyle*

hovered_checked
The "hovered_checked" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *CheckboxStyle*

pyiced.**ContainerStyle**
alias of *pyiced.ContainerStyleSheet*

class pyiced.**ContainerStyleSheet**(*proto=None*, ***kwargs*)
The appearance of a *container()*.

Parameters

- **proto** (*Optional[ContainerStyleSheet]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.
- **text_color** (*Optional[Color]*) – The container's text color.
- **background** (*Optional[Color]*) – The container's background color.
- **border_radius** (*float*) – The container's border radius.
- **border_width** (*float*) – The container's border width.
- **border_color** (*Color*) – The container's border color.

See also:

- iced::widget::container::Style
- iced::widget::container::StyleSheet

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type Optional[*Color*]

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

text_color
The "text_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type Optional[*Color*]

pyiced.**PaneGridStyle**
alias of *pyiced.PaneGridStyleSheet*

class pyiced.**PaneGridStyleSheet**(*proto=None*, ***kwargs*)
The appearance of a `pane_grid()`.

Parameters

- **proto** (*Optional[PaneGridStyleSheet]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.
- **picked_split** (*Optional[Line]*) – The line to draw when a split is picked.
- **hovered_split** (*Optional[Line]*) – The line to draw when a split is hovered.

See also:

- iced::widget::pane_grid::Style
- iced::widget::pane_grid::StyleSheet

hovered_split
The "hovered_split" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type Optional[*Line*]

picked_split
The "picked_split" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type Optional[*Line*]

class pyiced.**PickListMenu**(*proto=None*, ***kwargs*)
The appearance of a pick list menu.

Parameters

- **proto** (*Optional[PickListMenu]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.
- **text_color** (*Color*) – The text color of the menu.
- **background** (*Color*) – The background color of the menu.
- **border_width** (*float*) – The border width of the menu.
- **border_color** (*Color*) – The border color of the menu.

- **selected_text_color** (Color) – The text color of the selected element.
- **selected_background** (Color) – Text background color of the selected element.

See also:

iced::widget::pick_list::Menu

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

selected_background
The "selected_background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

selected_text_color
The "selected_text_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

text_color
The "text_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

class pyiced.**PickListStyle**(*proto=None*, ***kwargs*)
The appearance of a *pick_list()* for some state.

Parameters

- **proto** (*Optional[Union[PickListStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active" and "hovered", same as the argument for *PickListStyleSheet*.

  None is the same as "active".

- **text_color** (Color) – The pick list's foreground color.
- **background** (Color) – The pick list's background color.
- **border_radius** (*float*) – The pick list's border radius.

- **border_width** (*float*) – The pick list's border width.
- **border_color** (Color) – The pick list's border color.
- **icon_size** (*float*) – The pick list's arrow down icon size.

See also:

iced::widget::pick_list::Style

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

icon_size
The "icon_size" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

text_color
The "text_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

class pyiced.**PickListStyleSheet**(*menu*, *active*, *hovered=None*)
The appearance of a *pick_list()*.

Parameters

- **menu** (PickListMenu) – Style of the drop down menu.
- **active** (PickListStyle) – Normal style of the pick list.
- **hovered** (*Optional*[PickListStyle]) – Style of the pick list when the cursor is hovering over it. Defaults to "active".

See also:

iced::widget::pick_list::StyleSheet

active
The "active" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *PickListStyle*

hovered
The "hovered" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *PickListStyle*

menu
The "menu" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *PickListMenu*

pyiced.**ProgressBarStyle**
alias of *pyiced.ProgressBarStyleSheet*

class pyiced.**ProgressBarStyleSheet**(*proto=None*, ***kwargs*)
The appearance of a *progress_bar()*.

Parameters

- **background** (Color) – The progress bar's background color.
- **bar** (Color) – The progress bar's foreground color.
- **border_radius** (*float*) – The progress bar's border radius.

See also:

- iced::widget::progress_bar::Style
- iced::widget::progress_bar::StyleSheet

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

bar
The "bar" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

class pyiced.**RadioStyle**(*proto=None*, ***kwargs*)
The appearance of a *radio()* for some state.

Parameters

- **proto** (*Optional[Union[RadioStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active" and "hovered", same as the argument for *RadioStyleSheet*.

  None is the same as "active".

- **background** (*Color*) – The radio's background color.
- **dot_color** (*Color*) – The color of the dot.
- **border_width** (*float*) – The radio's border width.
- **border_color** (*Color*) – The radio's border color.

See also:

iced::widget::radio::Style

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

dot_color
The "dot_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

class pyiced.**RadioStyleSheet**(*active*, *hovered=None*)
The appearance of a *radio()*.

Parameters

- **active** (*RadioStyle*) – Normal style of the radio.
- **hovered** (*Optional[RadioStyle]*) – Style of the radio when the cursor is hovering over it. Defaults to "active".

See also:

iced::widget::radio::StyleSheet

active
The "active" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *RadioStyle*

hovered
The "hovered" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *RadioStyle*

pyiced.**RuleStyle**
alias of *pyiced.RuleStyleSheet*

class pyiced.**RuleStyleSheet**(*proto=None*, ***kwargs*)
The appearance of a *rule()*.

Parameters

- **proto** (*Optional[RuleStyleSheet]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.
- **color** (Color) – The color of the rule.
- **width** (*int*) – The width (thickness) of the rule line.
- **radius** (*float*) – The radius of the line corners.
- **fill_mode** (FillMode) – The fill mode of the rule.

See also:

- iced::widget::rule::Style
- iced::widget::rule::StyleSheet

color
The "color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

fill_mode
The "fill_mode" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *FillMode*

radius
The "radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

width
The "width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

class pyiced.**ScrollableStyleSheet**(*active*, *hovered=None*, *dragging=None*)
The appearance of a *scrollable()*.

Parameters

- **active** (ScrollbarStyle) – Normal style of the scrollable.

- **hovered** (*Optional[ScrollbarStyle]*) – Style of the scrollable when the cursor is hovering over it. Defaults to "active".
- **dragging** (*Optional[ScrollbarStyle]*) – Style of a scrollbar that is being dragged. Defaults to "hovered".

See also:

iced::widget::scrollable::StyleSheet

`active`
The "active" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ScrollbarStyle*

`dragging`
The "dragging" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ScrollbarStyle*

`hovered`
The "hovered" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ScrollbarStyle*

`class pyiced.ScrollbarStyle(proto=None, **kwargs)`
The appearance a specific state of a *scrollable()*.

Parameters

- **proto** (*Optional[Union[ScrollbarStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active", "hovered" and "dragging", same as the argument for *ScrollableStyleSheet*.

  None is the same as "active".
- **background** (*Optional[Color]*) – The scrollbar's background color.
- **border_radius** (*float*) – The scrollbar's border radius.
- **border_width** (*float*) – The scrollbar's border width.
- **border_color** (*Color*) – The scrollbar's border color.
- **scroller** (*ScrollerStyle*) – The scroller of the scrollbar.

See also:

iced_style::scrollable::Scrollbar

`background`
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type Optional[*Color*]

`border_color`
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

scroller
The "scroller" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *ScrollerStyle*

class pyiced.**ScrollerStyle**(*proto=None*, ***kwargs*)
The appearance of the scroller of a *scrollable()*.

Parameters

- **proto** (*Optional[Union[ScrollerStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active", "hovered" and "dragging", same as the argument for *ScrollableStyleSheet*.

  None is the same as "active".
- **color** (*Color*) – The color of the scroller.
- **border_radius** (*float*) – The border radius of the scroller.
- **border_width** (*float*) – The border width of the scroller.
- **border_color** (*Color*) – The border color of the scroller.

See also:

iced_style::scrollable::Scroller

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

`color`
The "color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

`class pyiced.SliderStyle(proto=None, **kwargs)`
The appearance of a *slider()* for some state.

Parameters

- **proto** (*Optional[Union[SliderStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active", "hovered" and "dragging", same as the argument for *SliderStyleSheet*.

  None is the same as "active".
- **rail_colors** (*Tuple[Color, Color]*) – TODO
- **handle** (SliderHandle) – TODO

See also:

iced::widget::slider::Style

`handle`
The "handle" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *SliderHandle*

`rail_colors`
The "rail_colors" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type Tuple[*Color*, *Color*]

`class pyiced.SliderStyleSheet(active, hovered=None, dragging=None)`
The appearance of a *slider()*.

Parameters

- **active** (SliderStyle) – Normal style of the slider.
- **hovered** (*Optional[SliderStyle]*) – Style of the slider when the cursor is hovering over it. Defaults to "active".
- **dragging** (*Optional[SliderStyle]*) – Style of the slider is being dragged. Defaults to "hovered".

See also:

iced::widget::slider::StyleSheet

`class pyiced.TextInputStyle(proto=None, **kwargs)`
The appearance of a *text_input()* for some state.

Parameters

- **proto** (*Optional[Union[TextInputStyle, str]]*) – Source style sheet to clone and modify. Defaults to iced_style's default style.

  The valid string values are "active", "focused" and "hovered", same as the argument for *TextInputStyleSheet*.

  None is the same as "active".

- **background** (*Color*) – The text_input's background color.
- **border_radius** (*float*) – The text_input's border radius.
- **border_width** (*float*) – The text_input's border width.
- **border_color** (*Color*) – The text_input's border color.

See also:

iced::widget::text_input::Style

background
The "background" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_color
The "border_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

border_radius
The "border_radius" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

border_width
The "border_width" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type float

class pyiced.**TextInputStyleSheet**(*active*, *focused=None*, *hovered=None*, *placeholder_color=None*, *value_color=None*, *selection_color=None*)
The appearance of a *text_input()*.

Parameters

- **active** (*TextInputStyle*) – Normal style of the text_input.
- **focused** (*Optional[TextInputStyle]*) – Style of the text_input when the cursor is hovering over it. Defaults to "active".
- **hovered** (*Optional[TextInputStyle]*) – Style of the text_input is being dragged. Defaults to "focused".
- **placeholder_color** (*Optional[Color]*) – Text color of the placeholder text.
- **value_color** (*Optional[Color]*) – Color of the text.
- **selection_color** (*Optional[Color]*) – Color of the selection.

See also:

iced::widget::text_input::StyleSheet

`active`
The "active" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *TextInputStyle*

`focused`
The "focused" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *TextInputStyle*

`hovered`
The "hovered" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *TextInputStyle*

`placeholder_color`
The "placeholder_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

`selection_color`
The "selection_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

`value_color`
The "value_color" parameter given to the constructor.

Returns The set, copied or defaulted value.

Return type *Color*

pyiced.**TooltipStyle**
alias of *pyiced.ContainerStyleSheet*

pyiced.**TooltipStyleSheet**
alias of *pyiced.ContainerStyleSheet*

4.9 Event Listening

4.9.1 Overview

every(duration, token)	Returns a *Subscription* that produces messages at a set interval.
stream(async_generator)	Listen for messages until the asynchronous generator is exhausted.
Subscription	TODO

4.9.2 Details

pyiced.**every**(*duration*, *token*)

Returns a *Subscription* that produces messages at a set interval.

The first *Message* is produced after a duration, and then continues to produce more messages every duration after that.

Parameters

- **duration** (*Union[float, datetime.timedelta]*) – The interval in seconds or as a duration. Must be at least 100 µs!
- **token** (*object*) – The first item of the message tuple to send to the *pyiced.IcedApp.update()*.

Returns

The new subscription.

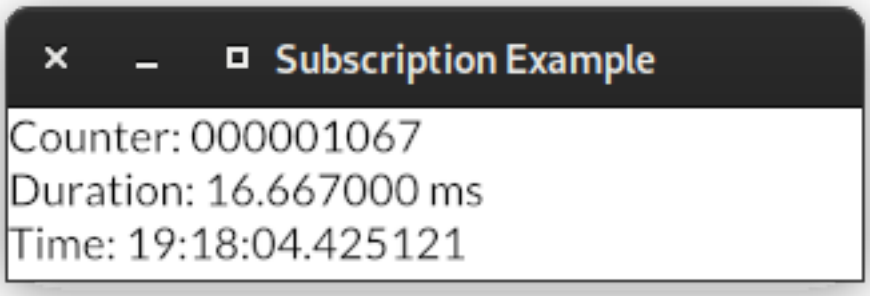
Every "duration" a message `(token, instant)` is sent to *pyiced.IcedApp.update()*.

See also:

Instant.

Return type *Subscription*

Example




```
from datetime import datetime, timedelta

from pyiced import column, every, IcedApp, Instant, text


class SubscriptionExample(IcedApp):
    def __init__(self):
        self.__counter = 0
        self.__instant = Instant()
        self.__last_instant = self.__instant
        self.__ts = datetime.now().time()
        self.__subscription = every(timedelta(milliseconds=16.667), 'tick')

    class settings:
```

(continues on next page)

(continued from previous page)

```
        class window:
            size = (320, 64)

    def title(self):
        return 'Subscription Example'

    def view(self):
        duration = self.__instant - self.__last_instant
        return column([
            text(f'Counter: {self.__counter:09d}'),
            text(f'Duration: {duration * 10**3:9.6f} ms'),
            text(f'Time: {self.__ts}')
        ])

    def subscriptions(self):
        return [self.__subscription]

    def update(self, msg, clipboard):
        match msg:
            case ('tick', instant):
                self.__last_instant = self.__instant
                self.__counter += 1
                self.__instant = instant
                self.__ts = datetime.now().time()


if __name__ == '__main__':
    SubscriptionExample().run()
```

See also:

iced_futures::time::every

pyiced.**stream**(*async_generator*)

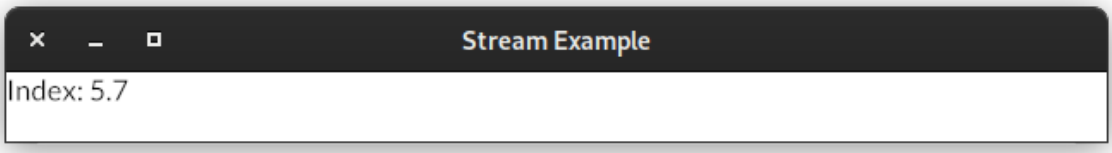
Listen for messages until the `asynchronous generator` is exhausted.

Parameters **async_generator** (*AsyncGenerator[Optional[object], None]*) – An asynchronous generator of messages.

Returns The wrapped generator.

Return type *Subscription*

Example



```python
from asyncio import sleep

from pyiced import column, IcedApp, stream, text


class StreamExample(IcedApp):
    def __init__(self):
        self.__stream = stream(self.__generator_func())
        self.__index = 0

    class settings:
        class window:
            size = (640, 40)

    def title(self):
        return 'Stream Example'

    def view(self):
        return column([text(f'Index: {self.__index / 10:.1f}')])

    def subscriptions(self):
        if self.__stream is not None:
            return [self.__stream]

    def update(self, msg, clipboard):
        match msg:
            case 'done':
                self.__stream = None
            case int(index):
                self.__index = index

    async def __generator_func(self):
        for i in range(1, 101):
            yield i
            await sleep(0.1)
        yield 'done'


if __name__ == '__main__':
    StreamExample().run()
```

See also:

iced_futures::subscription::Subscription

class pyiced.**Subscription**

TODO

NONE = <pyiced.Subscription object>

UNCAPTURED = <pyiced.Subscription object>

Glossary / Index

PYTHON MODULE INDEX

INDEX

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z