Curtsies Documentation

Release 0.2

Thomas Ballinger

Jun 06, 2017

Contents

1 Quickstart **3**

2 FmtStr **5**
- 2.1 FmtStr - Example . . . 5
- 2.2 FmtStr - Rationale . . . 5
- 2.3 FmtStr - Using . . . 6
  - 2.3.1 FmtStr - Using str methods . . . 7
  - 2.3.2 FmtStr - Unicode . . . 7
  - 2.3.3 FmtStr - len vs width . . . 8
- 2.4 FmtStr - API Docs . . . 8

3 FSArray **11**
- 3.1 FSArray - Example . . . 11
- 3.2 FSArray - Using . . . 11
- 3.3 FSArray - API docs . . . 12

4 Window Objects **15**
- 4.1 Window Objects - Example . . . 15
- 4.2 Window Objects - Context . . . 15
- 4.3 Window Objects - API Docs . . . 16

5 Input **19**
- 5.1 Input - Example . . . 19
- 5.2 Input - Getting Keyboard Events . . . 19
- 5.3 Input - Using as a Reactor . . . 19
- 5.4 Input - Context . . . 20
- 5.5 Input - Notes . . . 20
- 5.6 Input - Events . . . 20
  - 5.6.1 Input - Event Objects . . . 20
  - 5.6.2 Input - Keypress Strings . . . 21
- 5.7 Input - API docs . . . 21

6 Gameloop Example **23**

7 Examples **25**

8 About **27**

8.1 Resources 27
8.2 Authors 27

Python Module Index **29**

Curtsies is a Python 2.6+ & 3.3+ compatible library for interacting with the terminal.

FmtStr objects are strings formatted with colors and styles displayable in a terminal with ANSI escape sequences. *FSArray* objects contain multiple such strings with each formatted string on its own row, and can be superimposed onto each other to build complex grids of colored and styled characters.

Such grids of characters can be efficiently rendered to the terminal in alternate screen mode (no scrollback history, like `Vim`, `top` etc.) by *FullscreenWindow* objects or to the normal history-preserving screen by *CursorAwareWindow* objects. User keyboard input events like pressing the up arrow key are detected by an *Input* object. See the *Quickstart* to get started using all of these classes.

CHAPTER 1

Quickstart

This is what using (nearly every feature of) Curtsies looks like:

```
from __future__ import unicode_literals # convenient for Python 2
import random

from curtsies import FullscreenWindow, Input, FSArray
from curtsies.fmtfuncs import red, bold, green, on_blue, yellow

print(yellow('this prints normally, not to the alternate screen'))
with FullscreenWindow() as window:
    with Input() as input_generator:
        msg = red(on_blue(bold('Press escape to exit')))
        a = FSArray(window.height, window.width)
        a[0:1, 0:msg.width] = [msg]
        window.render_to_terminal(a)
        for c in input_generator:
            if c == '<ESC>':
                break
            elif c == '<SPACE>':
                a = FSArray(window.height, window.width)
            else:
                s = repr(c)
                row = random.choice(range(window.height))
                column = random.choice(range(window.width-len(s)))
                color = random.choice([red, green, on_blue, yellow])
                a[row, column:column+len(s)] = [color(s)]
            window.render_to_terminal(a)
```

Paste it into a file and try it out!

CHAPTER 2

FmtStr

FmtStr is a string with each character colored and styled in ways representable by ANSI escape codes.

FmtStr - Example

```
>>> from curtsies import fmtstr
>>> red_on_blue = fmtstr(u'hello', fg='red', bg='blue')
>>> from curtsies.fmtfuncs import *
>>> blue_on_red = blue(on_red(u'there'))
>>> bang = bold(underline(green(u'!')))
>>> full = red_on_blue + blue_on_red + bang
>>> str(full)

↪'\x1b[31m\x1b[44mhello\x1b[0m\x1b[0m\x1b[34m\x1b[41mthere\x1b[0m\x1b[0m\x1b[4m\x1b[32m\x1b[1m!
↪\x1b[0m\x1b[0m\x1b[0m'
>>> print(full)
hellothere!
```

We start here with such a complicated example because it you only need something simple like:

```
>>> from curtsies.fmtfuncs import *
>>> print(blue(bold(u'Deep blue sea')))
Deep blue sea
```

then another library may be a better fit than Curtsies. Unlike other libraries, Curtsies allows these colored strings to be further manipulated after they are created.

FmtStr - Rationale

If all you need is to print colored text, many other libraries also make ANSI escape codes easy to use.

- Blessings (`pip install blessings`) As of version 0.1.0, Curtsies uses Blessings for terminal capabilities other than colored output.
- termcolor (`pip install termcolor`)
- Clint (`pip install clint`)
- colors (`pip install colors`)

In all of the libraries listed above, the expression `blue('hi') + ' ' + green('there)` or equivalent evaluates to a Python string, not a colored string object. If all you plan to do with this string is print it, this is great. But, if you need to do more formatting with this colored string later, the length will be something like 29 instead of 9; structured formatting information is lost. Methods like `center` and `ljust` won't properly format the string for display.

```
>>> import blessings
>>> term = blessings.Terminal()
>>> message = term.red_on_green('Red on green?') + ' ' + term.yellow('Ick!')
>>> len(message)
41 # ?!
>>> message.center(50)
u'    \x1b[31m\x1b[42mRed on green?\x1b[m\x0f \x1b[33mIck!\x1b[m\x0f     '
```

FmtStr objects can be combined and composited to create more complicated *FmtStr* objects, useful for building flashy terminal interfaces with overlapping windows/widgets that can change size and depend on each other's sizes. One *FmtStr* can have several kinds of formatting applied to different parts of it.

```
>>> from curtsies.fmtfuncs import *
>>> blue('asdf') + on_red('adsf')
blue("asdf")+on_red("adsf")
```

FmtStr - Using

A *FmtStr* can be sliced to produce a new *FmtStr* objects:

```
>>> from curtsies.fmtfuncs import *
>>> (blue('asdf') + on_red('adsf'))[3:7]
blue("f")+on_red("ads")
```

FmtStr are *immutable* - but you can create new ones with *splice()*:

```
>>> from curtsies.fmtfuncs import *
>>> f = blue('hey there') + on_red(' Tom!')
>>> g.splice('ot', 1, 3)
>>> g
blue("h")+"ot"+blue(" there")+on_red(" Tom!")
>>> f.splice('something longer', 2)
blue("h")+"something longer"+blue("ot")+blue(" there")+on_red(" Tom!")
```

FmtStr greedily absorb strings, but no formatting is applied to this added text:

```
>>> from curtsies.fmtfuncs import *
>>> f = blue("The story so far:") + "In the beginning..."
>>> type(f)
<class curtsies.fmtstr.FmtStr>
>>> f
blue("The story so far:")+"In the beginning..."
```

It's easy to turn ANSI terminal formatted strings into *FmtStr*:

```
>>> from curtsies.fmtfuncs import *
>>> from curtsies import FmtStr
>>> s = str(blue('tom'))
>>> s
'\x1b[34mtom\x1b[39m'
>>> FmtStr.from_str(str(blue('tom')))
blue("tom")
```

FmtStr - Using str methods

All sorts of string methods can be used on a *FmtStr*, so you can often use *FmtStr* objects where you had strings in your program before:

```
>>> from curtsies.fmtfuncs import *
>>> f = blue(underline('As you like it'))
>>> len(f)
14
>>> f == underline(blue('As you like it')) + red('')
True
>>> blue(', ').join(['a', red('b')])
"a"+blue(", ")+red("b")
```

If *FmtStr* doesn't implement a method, it tries its best to use the string method, which often works pretty well:

```
>>> from curtsies.fmtfuncs import *
>>> f = blue(underline('As you like it'))
>>> f.center(20)
blue(underline("   As you like it   "))
>>> f.count('i')
2
>>> f.endswith('it')
True
>>> f.index('you')
3
>>> f.split(' ')
[blue(underline("As")), blue(underline("you")), blue(underline("like")),␣
↪blue(underline("it"))]
```

But formatting information will be lost for attributes which are not the same throughout the initial string:

```
>>> from curtsies.fmtfuncs import *
>>> f = bold(red('hi')+' '+on_blue('there'))
>>> f
bold(red('hi'))+bold(' ')+bold(on_blue('there'))
>>> f.center(10)
bold(" hi there ")
```

FmtStr - Unicode

In Python 2, you might run into something like this:

```
>>> from curtsies.fmtfuncs import *
>>> red(u'hi')
```

```
red('hi')
>>> red('hi')
ValueError: unicode string required, got 'hi'
```

FmtStr requires unicode strings, so in Python 2 it is convenient to use the unicode_literals compiler directive:

```
>>> from __future__ import unicode_literals
>>> from curtsies.fmtfuncs import *
>>> red('hi')
red('hi')
```

FmtStr - len vs width

The amount of horizontal space a string takes up in a terminal may differ from the length of the string returned by `len()`. *FmtStr* objects have a *width* property useful when writing layout code:

```
>>> #encoding: utf8
...
>>> from curtsies.fmtfuncs import *
>>> fullwidth = blue(u'width')
>>> len(fullwidth), fullwidth.width, fullwidth.s
(9, 13, u'\uff46\uff55\uff4c\uff4cwidth')
>>> combined = red(u'a')
>>> len(combined), combined.width, combined.s
(2, 1, u'a\u0324')
```

As shown above, full width characters can take up two columns, and combining characters may be combined with the previous character to form a single grapheme. Curtsies uses a Python implementation of wcwidth to do this calculation.

FmtStr - API Docs

curtsies.**fmtstr**(*string*, **args*, ***kwargs*)

Convenience function for creating a FmtStr

```
>>> fmtstr('asdf', 'blue', 'on_red', 'bold')
on_red(bold(blue('asdf')))
>>> fmtstr('blarg', fg='blue', bg='red', bold=True)
on_red(bold(blue('blarg')))
```

class curtsies.**FmtStr**(**components*)

A string whose substrings carry attributes (which may be different from one to the next).

copy_with_new_atts(***attributes*)

Returns a new FmtStr with the same content but new formatting

copy_with_new_str(*new_str*)

Copies the current FmtStr's attributes while changing its string.

join(*iterable*)

Joins an iterable yielding strings or FmtStrs with self as separator

splice(*new_str*, *start*, *end=None*)

Returns a new FmtStr with the input string spliced into the the original FmtStr at start and end. If end is provided, new_str will replace the substring self.s[start:end-1].

split(*sep=None*, *maxsplit=None*, *regex=False*)
Split based on seperator, optionally using a regex

Capture groups are ignored in regex, the whole pattern is matched and used to split the original FmtStr.

splitlines(*keepends=False*)
Return a list of lines, split on newline characters, include line boundaries, if keepends is true.

width
The number of columns it would take to display this string

width_aware_slice(*index*)
Slice based on the number of columns it would take to display the substring

FmtStr instances respond to most `str` methods as you might expect, but the result of these methods sometimes loses its formatting.

CHAPTER 3

FSArray

FSArray is a two dimensional grid of colored and styled characters.

FSArray - Example

```
>>> from curtsies import FSArray, fsarray
>>> from curtsies.fmtfuncs import green, blue, on_green
>>> a = fsarray([u'*' * 10 for _ in range(4)], bg='blue', fg='red')
>>> a.dumb_display()
**********
**********
**********
**********
>>> a[1:3, 3:7] = fsarray([green(u'msg:'),
...                        blue(on_green(u'hey!'))])
>>> a.dumb_display()
**********
***msg:***
***hey!***
**********
```

fsarray is a convenience function returning a *FSArray* constructed from its arguments.

FSArray - Using

FSArray objects can be composed to build up complex text interfaces:

```
>>> import time
>>> from curtsies import FSArray, fsarray, fmtstr
>>> def clock():
...     return fsarray([u'::'+fmtstr(u'time')+u'::',
```

```
...                     fmtstr(time.strftime('%H:%M:%S').decode('ascii'))])
...
>>> def square(width, height, char):
...     return fsarray(char*width for _ in range(height))
...
>>> a = square(40, 10, u'+')
>>> a[2:8, 2:38] = square(36, 6, u'.')
>>> c = clock()
>>> a[2:4, 30:38] = c
>>> a[6:8, 30:38] = c
>>> message = fmtstr(u'compositing several FSArrays').center(40, u'-')
>>> a[4:5, :] = [message]
>>>
>>> a.dumb_display()
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
++..............................::time::++
++..............................21:59:31++
------compositing several FSArrays------
++..................................++
++..............................::time::++
++..............................21:59:31++
++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++
```

An array like shown above might be repeatedly constructed and rendered with a *curtsies.window* object.

Slicing works like it does with a *FmtStr*, but in two dimensions. *FSArray* are *mutable*, so array assignment syntax can be used for natural compositing as in the above exaple.

If you're dealing with terminal output, the *width* of a string becomes more important than it's *length* (see *FmtStr - len vs width*).

In the future *FSArray* will do slicing and array assignment based on width instead of number of characters, but this is not currently implemented.

FSArray - API docs

curtsies.**fsarray**(*list_of_FmtStrs_or_strings*, *width=None*) → FSArray

Returns a new FSArray of width of the maximum size of the provided strings, or width provided, and height of the number of strings provided. If a width is provided, raises a ValueError if any of the strings are of length greater than this width

class curtsies.**FSArray**(*num_rows*, *num_columns*, **args*, ***kwargs*)

A 2D array of colored text.

Internally represented by a list of FmtStrs of identical size.

classmethod diff(*a*, *b*, *ignore_formatting=False*)

Returns two FSArrays with differences underlined

dumb_display()

Prints each row followed by a newline without regard for the terminal window size

height

The number of rows

shape
tuple of (len(rows, len(num_columns)) numpy-style shape

width
The number of columns

CHAPTER 4

Window Objects

Windows successively render 2D grids of text (usually instances of *FSArray*) to the terminal.

A window owns its output stream - it is assumed (but not enforced) that no additional data is written to this stream between renders, an assumption which allowing for example portions of the screen which do not change between renderings not to be redrawn during a rendering.

There are two useful window classes, both subclasses of *BaseWindow*. *FullscreenWindow* renders to the terminal's alternate screen buffer (no history preserved, like command line tools `Vim` and `top`) while *CursorAwareWindow* renders to the normal screen. It is also is capable of querying the terminal for the cursor location, and uses this functionality to detect how a terminal moves its contents around during a window size change. This information can be used to compensate for this movement and prevent the overwriting of history on the terminal screen.

Window Objects - Example

```
>>> from curtsies import FullscreenWindow, fsarray
>>> import time
>>> with FullscreenWindow() as win:
...     win.render_to_terminal(fsarray([u'asdf', u'asdf']))
...     time.sleep(1)
...     win.render_to_terminal(fsarray([u'asdf', u'qwer']))
...     time.sleep(1)
```

Window Objects - Context

`render_to_terminal()` should only be called within the context of a window. Within the context of an instance of *BaseWindow* it's important not to write to the stream the window is using (usually `sys.stdout`). Terminal window contents and even cursor position are assumed not to change between renders. Any change that does occur in cursor position is attributed to movement of content in response to a window size change and is used to calculate how this content has moved, necessary because this behavior differs between terminal emulators.

Entering the context of a *FullscreenWindow* object hides the cursor and switches to the alternate terminal screen. Entering the context of a *CursorAwareWindow* hides the cursor, turns on cbreak mode, and records the cursor position. Leaving the context does more or less the inverse.

Window Objects - API Docs

class curtsies.window.**BaseWindow**(*out_stream=None*, *hide_cursor=True*)

array_from_text(*msg*)
Returns a FSArray of the size of the window containing msg

get_term_hw()
Returns current terminal height and width

height
The current height of the terminal window

width
The current width of the terminal window

class curtsies.**FullscreenWindow**(*out_stream=None*, *hide_cursor=True*)
2D-text rendering window that dissappears when its context is left

FullscreenWindow will only render arrays the size of the terminal or smaller, and leaves no trace on exit (like top or vim). It never scrolls the terminal. Changing the terminal size doesn't do anything, but rendered arrays need to fit on the screen.

Note: The context of the FullscreenWindow object must be entered before calling any of its methods.

Within the context of CursorAwareWindow, refrain from writing to its out_stream; cached writes will be inaccurate.

Constructs a FullscreenWindow

Parameters

- **out_stream** (*file*) – Defaults to sys.__stdout__
- **hide_cursor** (*bool*) – Hides cursor while in context

render_to_terminal(*array*, *cursor_pos=(0, 0)*)
Renders array to terminal and places (0-indexed) cursor

Parameters **array** (FSArray) – Grid of styled characters to be rendered.

•If array received is of width too small, render it anyway

•If array received is of width too large,

•render the renderable portion

•If array received is of height too small, render it anyway

•If array received is of height too large,

•render the renderable portion (no scroll)

class curtsies.**CursorAwareWindow**(*out_stream=None*, *in_stream=None*, *keep_last_line=False*, *hide_cursor=True*, *extra_bytes_callback=None*)

Renders to the normal terminal screen and can find the location of the cursor.

Note: The context of the CursorAwareWindow object must be entered before calling any of its methods.

Within the context of CursorAwareWindow, refrain from writing to its out_stream; cached writes will be inaccurate and calculating cursor depends on cursor not having moved since the last render. Only use the render_to_terminal interface for moving the cursor.

Constructs a CursorAwareWindow

Parameters

- **out_stream** (*file*) – Defaults to sys.__stdout__
- **in_stream** (*file*) – Defaults to sys.__stdin__
- **keep_last_line** (*bool*) – Causes the cursor to be moved down one line on leaving context
- **hide_cursor** (*bool*) – Hides cursor while in context
- **extra_bytes_callback** (*f(bytes) -> None*) – Will be called with extra bytes inadvertantly read in get_cursor_position(). If not provided, a ValueError will be raised when this occurs.

get_cursor_position()

Returns the terminal (row, column) of the cursor

0-indexed, like blessings cursor positions

get_cursor_vertical_diff()

Returns the how far down the cursor moved since last render.

Note: If another get_cursor_vertical_diff call is already in progress, immediately returns zero. (This situation is likely if get_cursor_vertical_diff is called from a SIGWINCH signal handler, since sigwinches can happen in rapid succession and terminal emulators seem not to respond to cursor position queries before the next sigwinch occurs.)

---

render_to_terminal(*array*, *cursor_pos=(0, 0)*)

Renders array to terminal, returns the number of lines scrolled offscreen

Returns Number of times scrolled

Parameters **array** (FSArray) – Grid of styled characters to be rendered.

If array received is of width too small, render it anyway

if array received is of width too large, render it anyway

if array received is of height too small, render it anyway

if array received is of height too large, render it, scroll down, and render the rest of it, then return how much we scrolled down

CHAPTER 5

Input

Input objects provide user keypress events and other control events.

Input - Example

```
>>> from curtsies import Input
>>> with Input(keynames='curtsies') as input_generator:
...     for e in Input():
...         if e in (u'<ESC>', u'<Ctrl-d>'):
...             break
...         else:
...             print(e)
```

Input - Getting Keyboard Events

The simplest way to use an *Input* object is to iterate over it in a for loop: each time a keypress is detected or other event occurs, an event is produced and can be acted upon. Since it's iterable, `next()` can be used to wait for a single event. *send()* works like `next()` but takes a timeout in seconds, which if reached will cause None to be returned signalling that no keypress or other event occured within the timeout.

Key events are unicode strings, but sometimes event objects (see *Event*) are returned instead. Built-in events signal *SigIntEvent* events from the OS and *PasteEvent* consisting of multiple keypress events if reporting of these types of events was enabled in instantiation of the *Input* object.

Input - Using as a Reactor

Custom events can also be scheduled to be returned from *Input* with callback functions created by the event trigger methods.

Each of these methods returns a callback that will schedule an instance of the desired event type:

- Using a callback created by *event_trigger()* schedules an event to be returned the next time an event is requested, but not if an event has already been requested (if called from another thread).
- *threadsafe_event_trigger()* does the same, but may notify a concurrent request for an event so that the custom event is immediately returned.
- *scheduled_event_trigger()* schedules an event to be returned at some point in the future.

Input - Context

`next()` and *send()* must be used within the context of that *Input* object.

Within the (context-manager) context of an Input generator, an in-stream is put in raw mode or cbreak mode, and keypresses are stored to be reported later. Original tty attributes are recorded to be restored on exiting the context. The SigInt signal handler may be replaced if this behavior was specified on creation of the *Input* object.

Input - Notes

Input takes an optional argument `keynames` for how to name keypress events, which is `'curtsies'` by default. For compatibility with curses code, you can use `'curses'` names, but note that curses doesn't have nice key names for many key combinations so you'll be putting up with names like `u'\xe1'` for `option-j` and `'\x86'` for `ctrl-option-f`. Pass `'plain'` for this parameter to return a simple unicode representation.

PasteEvent objects representing multiple keystrokes in very rapid succession (typically because the user pasted in text, but possibly because they typed two keys simultaneously). How many bytes must occur together to trigger such an event is customizable via the `paste_threshold` argument to the *Input* object - by default it's one greater than the maximum possible keypress length in bytes.

If `sigint_event=True` is passed to *Input*, `SIGINT` signals from the operating system (which usually raise a `KeyboardInterrupt` exception) will be returned as *SigIntEvent* instances.

To set a timeout on the blocking get, treat it like a generator and call `.send(timeout)`. The call will return `None` if no event is available.

Input - Events

To see what a given keypress is called (what unicode string is returned by `Terminal.next()`), try `python -m curtsies.events` and play around. Events returned by *Input* fall into two categories: instances of subclasses of *Event* and Keypress strings.

Input - Event Objects

class `curtsies.events.`**Event**

class `curtsies.events.`**SigIntEvent**
Event signifying a SIGINT

class `curtsies.events.`**PasteEvent**
Multiple keypress events combined, likely from copy/paste.

The events attribute contains a list of keypress event strings.

class curtsies.events.**ScheduledEvent**(*when*)
Event scheduled for a future time.

Parameters **when** (*float*) – unix time in seconds for which this event is scheduled

Custom events that occur at a specific time in the future should be subclassed from ScheduledEvent.

Input - Keypress Strings

Keypress events are Unicode strings in both Python 2 and 3 like:

- a, 4, *, ?
- <UP>, <DOWN>, <RIGHT>, <LEFT>
- <SPACE>, <TAB>, <F1>, <F12>
- <BACKSPACE>, <HOME>, <PADENTER>, <PADDELETE>
- <Ctrl+a>, <Ctrl+SPACE>
- A, <Shift-TAB>, ?
- <Esc+a>, <Esc+A>, <Esc+Ctrl-A>
- <Esc+Ctrl+A>
- <Meta-J>, <Meta-Ctrl-J> (this is old-style meta)

Likely points of confusion for keypress strings:

- Enter is <Ctrl-j>
- Modern meta (the escape-prepending version) key is <Esc+a> while control and shift keys are <Ctrl-a> (note the + vs -)
- Letter keys are capitalized in <Esc+Ctrl-A> while they are lowercase in <Ctrl-a> (this should be fixed in the next api-breaking release)
- Some special characters lose their special names when used with modifier keys, for example: <TAB>, <Shift-TAB>, <Esc+Ctrl-Y>, <Esc+Ctrl-I> are all produced by the tab key, while y, Y, <Shift-TAB>, <Esc+y>, <Esc+Y>, <Esc+Ctrl-y>, <Esc+Ctrl-Y>, <Ctrl-Y> are all produced by the y key. (This should really be figured out)

Input - API docs

class curtsies.**Input**(*in_stream=None*, *keynames='curtsies'*, *paste_threshold=8*, *sigint_event=False*)
Keypress and control event generator

Returns an Input instance.

Parameters

- **in_stream** (*file*) – Defaults to sys.__stdin__
- **keynames** (*string*) – How keypresses should be named - one of 'curtsies', 'curses', or 'plain'.
- **paste_threshold** (*int*) – How many bytes must be read in one os.read on the in_stream to trigger the keypresses they represent to be combined into a single paste event

- **sigint_event** (*bool*) – Whether SIGINT signals from the OS should be intercepted and returned as SigIntEvent objects

unget_bytes(*string*)
Adds bytes to be internal buffer to be read

This method is for reporting bytes from an in_stream read not initiated by this Input object

send(*timeout=None*)
Returns an event or None if no events occur before timeout.

event_trigger(*event_type*)
Returns a callback that creates events.

Returned callback function will add an event of type event_type to a queue which will be checked the next time an event is requested.

scheduled_event_trigger(*event_type*)
Returns a callback that schedules events for the future.

Returned callback function will add an event of type event_type to a queue which will be checked the next time an event is requested.

threadsafe_event_trigger(*event_type*)
Returns a callback to creates events, interrupting current event requests.

Returned callback function will create an event of type event_type which will interrupt an event request if one is concurrently occuring, otherwise adding the event to a queue that will be checked on the next event request.

CHAPTER 6

Gameloop Example

Use scheduled events for realtime interative programs:

```
from __future__ import unicode_literals, division

import time

from curtsies import FullscreenWindow, Input, FSArray
from curtsies.fmtfuncs import red, bold, green, on_blue, yellow, on_red
import curtsies.events

class Frame(curtsies.events.ScheduledEvent):
    pass

class World(object):
    def __init__(self):
        self.s = 'Hello'
    def tick(self):
        self.s += '|'
        self.s = self.s[max(1, len(self.s)-80):]
    def process_event(self, e):
        self.s += str(e)

def realtime(fps=15):
    world = World()
    dt = 1/fps

    reactor = Input()
    schedule_next_frame = reactor.scheduled_event_trigger(Frame)
    schedule_next_frame(when=time.time())

    with reactor:
        for e in reactor:
            if isinstance(e, Frame):
                world.tick()
                print(world.s)
```

```
                when = e.when + dt
                while when < time.time():
                    when += dt
                schedule_next_frame(when)
            elif e == u'<ESC>':
                break
            else:
                world.process_event(e)

realtime()
```

Paste it into a file and try it out!

CHAPTER 7

Examples

- Tic-Tac-Toe
- Avoid the X's game
- Bpython-curtsies uses curtsies

CHAPTER 8

About

Resources

I've written a little bit about Curtsies on my blog.

The source and issue tracker for Curtsies are on Github.

A good place to ask questions about Curtsies is #bpython on irc.freenode.net.

Authors

Curtsies was written by Thomas Ballinger to create a frontend for bpython that preserved terminal history.

Thanks so much to the many people that have contributed to it!

- Amber Wilcox-O'Hearn - paired on a refactoring
- Darius Bacon - lots of great code review
- Fei Dong - work on making FmtStr and Chunk immutable
- Julia Evans - help with Python 3 compatibility
- Lea Albaugh - beautiful Curtsies logo
- Rachel King - several bugfixes on blessings use
- Scott Feeney - inspiration for this project - the original title of the project was "scott was right"
- Zach Allaun, Mary Rose Cook, Alex Clemmer - early code review of input and window
- Chase Lambert - API redesign conversation

Python Module Index

C

curtsies.fmtfuncs, 9
curtsies.formatstring, 5
curtsies.window, 15

Index

A

array_from_text() (curtsies.window.BaseWindow method), 16

B

BaseWindow (class in curtsies.window), 16

C

copy_with_new_atts() (curtsies.FmtStr method), 8
copy_with_new_str() (curtsies.FmtStr method), 8
CursorAwareWindow (class in curtsies), 16
curtsies.fmtfuncs (module), 9
curtsies.formatstring (module), 5
curtsies.window (module), 15

D

diff() (curtsies.FSArray class method), 12
dumb_display() (curtsies.FSArray method), 12

E

Event (class in curtsies.events), 20
event_trigger() (curtsies.Input method), 22

F

FmtStr (class in curtsies), 8
fmtstr() (in module curtsies), 8
FSArray (class in curtsies), 12
fsarray() (in module curtsies), 12
FullscreenWindow (class in curtsies), 16

G

get_cursor_position() (curtsies.CursorAwareWindow method), 17
get_cursor_vertical_diff() (curtsies.CursorAwareWindow method), 17
get_term_hw() (curtsies.window.BaseWindow method), 16

H

height (curtsies.FSArray attribute), 12
height (curtsies.window.BaseWindow attribute), 16

I

Input (class in curtsies), 21

J

join() (curtsies.FmtStr method), 8

P

PasteEvent (class in curtsies.events), 20

R

render_to_terminal() (curtsies.CursorAwareWindow method), 17
render_to_terminal() (curtsies.FullscreenWindow method), 16

S

scheduled_event_trigger() (curtsies.Input method), 22
ScheduledEvent (class in curtsies.events), 20
send() (curtsies.Input method), 22
shape (curtsies.FSArray attribute), 12
SigIntEvent (class in curtsies.events), 20
splice() (curtsies.FmtStr method), 8
split() (curtsies.FmtStr method), 8
splitlines() (curtsies.FmtStr method), 9

T

threadsafe_event_trigger() (curtsies.Input method), 22

U

unget_bytes() (curtsies.Input method), 22

W

width (curtsies.FmtStr attribute), 9
width (curtsies.FSArray attribute), 13
width (curtsies.window.BaseWindow attribute), 16
width_aware_slice() (curtsies.FmtStr method), 9