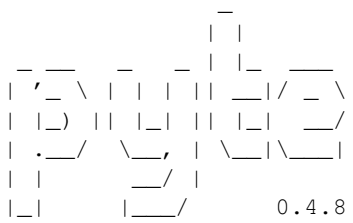
pyte

Release 0.4.8

January 13, 2014



It's an in memory VTXXX-compatible terminal emulator. *XXX* stands for a series of video terminals, developed by DEC between 1970 and 1995. The first, and probably the most famous one, was VT100 terminal, which is now a de-facto standard for all virtual terminal emulators. `pyte` follows the suit.

So, why would one need a terminal emulator library?

- To screen scrape terminal apps, for example `htop` or `aptitude`.
- To write cross platform terminal emulators; either with a graphical (xterm, rxvt) or a web interface, like Ajax-Term.
- To have fun, hacking on the ancient, poorly documented technologies.

Note: `pyte` started as a fork of vt102, which is an incomplete pure Python implementation of VT100 terminal.

CHAPTER 1

Installation

If you have setuptools you can use `easy_install -U pyte`. Otherwise, you can download the source from GitHub and run `python setup.py install`.

CHAPTER 2

Similar projects

`pyte` is not alone in the weird world of terminal emulator libraries, here's a few other options worth checking out: Termemulator, pyqonsole, webtty, AjaxTerm and of course vt102.

CHAPTER 3

`pyte` users

Believe it or not, there're projects which actually **need** a terminal emulator library, not many of them use `pyte` though:

- Selectel – hey, we wrote this thing :)
- Ajenti – easy to use weadmin panel for Linux and BSD uses `pyte` for its terminal plugin.
- Neutron IDE – personal cloud IDE.

CHAPTER 4

Show me the code!

Head over to our brief *Tutorial* or, if you're feeling brave, dive right into the *API reference* – `pyte` also has a couple of minimal examples in the `examples/` directory.

4.1 Tutorial

There are two important classes in `pyte`: `Screen` and `Stream`. The *Screen* is the terminal screen emulator. It maintains an in-memory buffer of text and text-attributes to display on screen. The *Stream* is the stream processor. It manages the state of the input and dispatches events to anything that's listening about things that are going on. Events are things like `LINEFEED`, `DRAW "a"`, or `CURSOR_POSITION 10 10`. See the *API reference* for more details.

In general, if you just want to know what's being displayed on screen you can do something like the following:

```
>>> import pyte
>>> screen = pyte.Screen(80, 24)
>>> stream = pyte.Stream()
>>> stream.attach(screen)
>>> stream.feed("Hello World!")
>>> screen.display
    ['Hello World!                                                                  ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ',
     '                                                                              ']
```

```
    ',                                                                                ',
    '                                                                                 ']
```

Note: `Screen` has no idea what is the source of bytes, fed into `Stream`, so, obviously, it **can't read** or **change** environment variables, which implies that:

- it doesn't adjust *LINES* and *COLUMNS* on `"resize"` event;
- it doesn't use locale settings (*LC_** and *LANG*);
- it doesn't use *TERM* value and expects it to be *"linux"* and only *"linux"*.

4.2 API reference

4.2.1 pyte

pyte implements a mix of VT100, VT220 and VT520 specification, and aims to support most of the *TERM=linux* functionality.

Two classes: `Stream`, which parses the command stream and dispatches events for commands, and `Screen` which, when used with a stream maintains a buffer of strings representing the screen of a terminal.

Warning: From `xterm/main.c` "If you think you know what all of this code is doing, you are probably very mistaken. There be serious and nasty dragons here" – nothing has changed.

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

4.2.2 pyte.streams

This module provides three stream implementations with different features; for starters, here's a quick example of how streams are typically used:

```
>>> import pyte
>>>
>>> class Dummy(object):
...     def __init__(self):
...         self.y = 0
...
...     def cursor_up(self, count=None):
...         self.y += count or 1
...
>>> dummy = Dummy()
>>> stream = pyte.Stream()
>>> stream.attach(dummy)
>>> stream.feed(u"[5A")  # Move the cursor up 5 rows.
>>> dummy.y
5
```

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

class pyte.streams.**Stream**

A stream is a state machine that parses a stream of characters and dispatches events based on what it sees.

Note: Stream only accepts strings as input, but if, for some reason, you need to feed it with bytes, consider using `ByteStream` instead.

See also:

man console_codes For details on console codes listed bellow in `basic`, `escape`, `csi` and `sharp`.

basic = {u'\x07': u'bell', u'\t': u'tab', u'\x08': u'backspace', u'\x0b': u'linefeed', u'\n': u'linefeed', u'\r': u'carriage_re

Control sequences, which don't require any arguments.

escape = {u'c': u'reset', u'E': u'linefeed', u'D': u'index', u'H': u'set_tab_stop', u'M': u'reverse_index', u'7': u'save_cu

non-CSI escape sequences.

sharp = {u'8': u'alignment_display'}

"sharp" escape sequences – `ESC # <N>`.

csi = {u'"': u'cursor_to_column', u'A': u'cursor_up', u'@': u'insert_characters', u'C': u'cursor_forward', u'B': u'curs

CSI escape sequences – `CSI P1;P2;...;Pn <fn>`.

reset()

Reset state to `"stream"` and empty parameter attributes.

consume(*char*)

Consume a single string character and advance the state as necessary.

Parameters **char** (*str*) – a character to consume.

feed(*chars*)

Consume a string and advance the state as necessary.

Parameters **chars** (*str*) – a string to feed from.

attach(*screen*, *only=()*)

Adds a given screen to the listeners queue.

Parameters

- **screen** (*pyte.screens.Screen*) – a screen to attach to.
- **only** (*list*) – a list of events you want to dispatch to a given screen (empty by default, which means – dispatch all events).

detach(*screen*)

Removes a given screen from the listeners queue and failes silently if it's not attached.

Parameters **screen** (*pyte.screens.Screen*) – a screen to detach.

dispatch(*event*, **args*, ***kwargs*)

Dispatch an event.

Event handlers are looked up implicitly in the listeners' `__dict__`, so, if a listener only wants to handle `DRAW` events it should define a `draw()` method or pass `only=["draw"]` argument to `attach()`.

Warning: If any of the attached listeners throws an exception, the subsequent callbacks are be aborted.

Parameters

- **event** (*str*) – event to dispatch.

- **args** (*list*) – arguments to pass to event handlers.

class pyte.streams.**ByteStream**(*encodings=None*)

A stream, which takes bytes (instead of strings) as input and tries to decode them using a given list of possible encodings. It uses codecs.IncrementalDecoder internally, so broken bytes is not an issue.

By default, the following decoding strategy is used:

- First, try strict "utf-8", proceed if recieved and UnicodeDecodeError ...
- Try strict "cp437", failed? move on ...
- Use "utf-8" with invalid bytes replaced – this one will allways succeed.

```
>>> stream = ByteStream()
>>> stream.feed(b"foo".decode("utf-8"))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pyte/streams.py", line 323, in feed
    "%s requires input in bytes" % self.__class__.__name__)
TypeError: ByteStream requires input in bytes
>>> stream.feed(b"foo")
```

Parameters **encodings** (*list*) – a list of (encoding, errors) pairs, where the first element is encoding name, ex: "utf-8" and second defines how decoding errors should be handeld; see str.decode() for possible values.

class pyte.streams.**DebugStream**(*to=<open file '<stdout>', mode 'w' at 0x7f3e3d4b9150>, only=(), *args, **kwargs*)

Stream, which dumps a subset of the dispatched events to a given file-like object (sys.stdout by default).

```
>>> stream = DebugStream()
>>> stream.feed("[1;24r[4l[24;1H[0;10m")
SET_MARGINS 1; 24
RESET_MODE 4
CURSOR_POSITION 24; 1
SELECT_GRAPHIC_RENDITION 0; 10
```

Parameters

- **to** (*file*) – a file-like object to write debug information to.
- **only** (*list*) – a list of events you want to debug (empty by default, which means – debug all events).

4.2.3 pyte.screens

This module provides classes for terminal screens, currently it contains three screens with different features:

- Screen – base screen implementation, which handles all the core escape sequences, recognized by Stream.
- If you need a screen to keep track of the changed lines (which you probably do need) – use DiffScreen.
- If you also want a screen to collect history and allow pagination – pyte.screen.HistoryScreen is here for ya ;)

Note: It would be nice to split those features into mixin classes, rather than subclasses, but it's not obvious how to do – feel free to submit a pull request.

---

copyright

3. 2011-2013 Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

class `pyte.screens.`**`Cursor`**(*x*, *y*, *attrs=_Char(data=u' ', fg=u'default', bg=u'default', bold=False, italics=False, underscore=False, strikethrough=False, reverse=False)*)

Screen cursor.

Parameters

- **x** (*int*) – horizontal cursor position.
- **y** (*int*) – vertical cursor position.
- **attrs** (*pyte.screens.Char*) – cursor attributes (see `selectel_graphic_rendition()` for details).

class `pyte.screens.`**`Screen`**(*columns*, *lines*)

A screen is an in-memory matrix of characters that represents the screen display of the terminal. It can be instantiated on it's own and given explicit commands, or it can be attached to a stream and will respond to events.

`buffer`

A `lines x columns` `Char` matrix.

`cursor`

Reference to the `Cursor` object, holding cursor position and attributes.

`margins`

Top and bottom screen margins, defining the scrolling region; the actual values are top and bottom line.

`charset`

Current charset number; can be either `0` or `1` for *G0* and *G1* respectively, note that *G0* is activated by default.

Note: According to `ECMA-48` standard, **lines and columnns are 1-indexed**, so, for instance `ESC [ 10;10 f` really means – move cursor to position (9, 9) in the display matrix.

Changed in version 0.4.7.

> **Warning:** `LNM` is reset by default, to match VT220 specification.

Changed in version 0.4.8.

> **Warning:** If *DECAWM* mode is set than a cursor will be wrapped to the **beginning* of the next line, which is the behaviour described in `man console_codes`.

See also:

Standard ECMA-48, Section 6.1.1

For a description of the presentational component, implemented by `Screen`.

`default_char` = _Char(data=u' ', fg=u'default', bg=u'default', bold=False, italics=False, underscore=False, strikethrou

A plain empty character with default foreground and background colors.

`default_line` = repeat(_Char(data=u' ', fg=u'default', bg=u'default', bold=False, italics=False, underscore=False, stri

An inifinite sequence of default characters, used for populating new lines and columns.

size
Returns screen size – `(lines, columns)`

display
Returns a `list()` of screen lines as unicode strings.

reset`()`
Resets the terminal to its initial state.

- Scroll margins are reset to screen boundaries.
- Cursor is moved to home location – `(0, 0)` and its attributes are set to defaults (see `default_char`).
- Screen is cleared – each character is reset to `default_char`.
- Tabstops are reset to "every eight columns".

Note: Neither VT220 nor VT102 manuals mentioned that terminal modes and tabstops should be reset as well, thanks to *`xterm`* – we now know that.

---

resize(*lines=None*, *columns=None*)
Resize the screen to the given dimensions.

If the requested screen size has more lines than the existing screen, lines will be added at the bottom. If the requested size has less lines than the existing screen lines will be clipped at the top of the screen. Similarly, if the existing screen has less columns than the requested screen, columns will be added at the right, and if it has more – columns will be clipped at the right.

Note: According to *xterm*, we should also reset origin mode and screen margins, see `xterm/screen.c:1761`.

Parameters

- **lines** (*int*) – number of lines in the new screen.
- **columns** (*int*) – number of columns in the new screen.

set_margins(*top=None*, *bottom=None*)
Selects top and bottom margins for the scrolling region.

Margins determine which screen lines move during scrolling (see `index()` and `reverse_index()`). Characters added outside the scrolling region do not cause the screen to scroll.

Parameters

- **top** (*int*) – the smallest line number that is scrolled.
- **bottom** (*int*) – the biggest line number that is scrolled.

set_charset(*code*, *mode*)
Set active `G0` or `G1` charset.

Parameters

- **code** (*str*) – character set code, should be a character from `"B0UK"` – otherwise ignored.
- **mode** (*str*) – if `"("` `G0` charset is set, if `")"` – we operate on `G1`.

Warning: User-defined charsets are currently not supported.

set_mode(**modes*, ***kwargs*)
Sets (enables) a given list of modes.

Parameters **modes** (*list*) – modes to set, where each mode is a constant from `pyte.modes`.

reset_mode(**modes*, ***kwargs*)
Resets (disables) a given list of modes.

Parameters **modes** (*list*) – modes to reset – hopefully, each mode is a constant from `pyte.modes`.

shift_in()
Activates `G0` character set.

shift_out()
Activates `G1` character set.

draw(*char*)
Display a character at the current cursor position and advance the cursor if `DECAWM` is set.

Parameters **char** (*str*) – a character to display.

carriage_return()
Move the cursor to the beginning of the current line.

index()
Move the cursor down one line in the same column. If the cursor is at the last line, create a new line at the bottom.

reverse_index()
Move the cursor up one line in the same column. If the cursor is at the first line, create a new line at the top.

linefeed()
Performs an index and, if `LNM` is set, a carriage return.

tab()
Move to the next tab space, or the end of the screen if there aren't anymore left.

backspace()
Move cursor to the left one or keep it in it's position if it's at the beginning of the line already.

save_cursor()
Push the current cursor position onto the stack.

restore_cursor()
Set the current cursor position to whatever cursor is on top of the stack.

insert_lines(*count=None*)
Inserts the indicated # of lines at line with cursor. Lines displayed **at** and below the cursor move down. Lines moved past the bottom margin are lost.

Parameters **count** – number of lines to delete.

delete_lines(*count=None*)
Deletes the indicated # of lines, starting at line with cursor. As lines are deleted, lines displayed below cursor move up. Lines added to bottom of screen have spaces with same character attributes as last line moved up.

Parameters **count** (*int*) – number of lines to delete.

insert_characters(*count=None*)
Inserts the indicated # of blank characters at the cursor position. The cursor does not move and remains at the beginning of the inserted blank characters. Data on the line is shifted forward.

Parameters **count** (*int*) – number of characters to insert.

delete_characters(*count=None*)
Deletes the indicated # of characters, starting with the character at cursor position. When a character is deleted, all characters to the right of cursor move left. Character attributes move with the characters.

Parameters **count** (*int*) – number of characters to delete.

erase_characters(*count=None*)
Erases the indicated # of characters, starting with the character at cursor position. Character attributes are set cursor attributes. The cursor remains in the same position.

Parameters **count** (*int*) – number of characters to erase.

> **Warning:** Even though *ALL* of the VTXXX manuals state that character attributes **should be reset to defaults**, `libvte`, `xterm` and `ROTE` completely ignore this. Same applies too all `erase_*()` and `delete_*()` methods.

erase_in_line(*type_of=0*, *private=False*)
Erases a line in a specific way.

Parameters

- **type_of** (*int*) – defines the way the line should be erased in:
  - `0` – Erases from cursor to end of line, including cursor position.
  - `1` – Erases from beginning of line to cursor, including cursor position.
  - `2` – Erases complete line.
- **private** (*bool*) – when `True` character attributes aren left unchanged **not implemented**.

erase_in_display(*type_of=0*, *private=False*)
Erases display in a specific way.

Parameters

- **type_of** (*int*) – defines the way the line should be erased in:
  - `0` – Erases from cursor to end of screen, including cursor position.
  - `1` – Erases from beginning of screen to cursor, including cursor position.
  - `2` – Erases complete display. All lines are erased and changed to single-width. Cursor does not move.
- **private** (*bool*) – when `True` character attributes aren left unchanged **not implemented**.

set_tab_stop()
Sest a horizontal tab stop at cursor position.

clear_tab_stop(*type_of=None*)
Clears a horizontal tab stop in a specific way, depending on the `type_of` value:

- `0` or nothing – Clears a horizontal tab stop at cursor position.
- `3` – Clears all horizontal tab stops.

ensure_bounds(*use_margins=None*)
Ensure that current cursor position is within screen bounds.

Parameters **use_margins** (*bool*) – when `True` or when `DECOM` is set, cursor is bounded by top and and bottom margins, instead of `[0; lines - 1]`.

cursor_up(*count=None*)
Moves cursor up the indicated # of lines in same column. Cursor stops at top margin.

Parameters **count** (*int*) – number of lines to skip.

cursor_up1(*count=None*)
Moves cursor up the indicated # of lines to column 1. Cursor stops at bottom margin.

Parameters **count** (*int*) – number of lines to skip.

cursor_down(*count=None*)
Moves cursor down the indicated # of lines in same column. Cursor stops at bottom margin.

Parameters **count** (*int*) – number of lines to skip.

cursor_down1(*count=None*)
Moves cursor down the indicated # of lines to column 1. Cursor stops at bottom margin.

Parameters **count** (*int*) – number of lines to skip.

cursor_back(*count=None*)
Moves cursor left the indicated # of columns. Cursor stops at left margin.

Parameters **count** (*int*) – number of columns to skip.

cursor_forward(*count=None*)
Moves cursor right the indicated # of columns. Cursor stops at right margin.

Parameters **count** (*int*) – number of columns to skip.

cursor_position(*line=None*, *column=None*)
Set the cursor to a specific *line* and *column*.

Cursor is allowed to move out of the scrolling region only when `DECOM` is reset, otherwise – the position doesn't change.

Parameters

- **line** (*int*) – line number to move the cursor to.
- **column** (*int*) – column number to move the cursor to.

cursor_to_column(*column=None*)
Moves cursor to a specific column in the current line.

Parameters **column** (*int*) – column number to move the cursor to.

cursor_to_line(*line=None*)
Moves cursor to a specific line in the current column.

Parameters **line** (*int*) – line number to move the cursor to.

bell(**args*)
Bell stub – the actual implementation should probably be provided by the end-user.

alignment_display()
Fills screen with uppercase E's for screen focus and alignment.

select_graphic_rendition(**attrs*)
Set display attributes.

Parameters **attrs** (*list*) – a list of display attributes to set.

class `pyte.screens.`**DiffScreen**(**args*)
A screen subclass, which maintains a set of dirty lines in its `dirty` attribute. The end user is responsible for emptying a set, when a diff is applied.

dirty
A set of line numbers, which should be re-drawn.

```
>>> screen = DiffScreen(80, 24)
>>> screen.dirty.clear()
>>> screen.draw(u"!")
>>> screen.dirty
set([0])
```

class pyte.screens.**HistoryScreen**(*columns*, *lines*, *history=100*, *ratio=0.5*)
A screen subclass, which keeps track of screen history and allows pagination. This is not linux-specific, but still useful; see page 462 of VT520 User's Manual.

Parameters

- **history** (*int*) – total number of history lines to keep; is split between top and bottom queues.
- **ratio** (*int*) – defines how much lines to scroll on `next_page()` and `prev_page()` calls.

history
A pair of history queues for top and bottom margins accordingly; here's the overall screen structure:

```
[ 1: .......]
[ 2: .......]  <- top history
[ 3: .......]
------------
[ 4: .......]  s
[ 5: .......]  c
[ 6: .......]  r
[ 7: .......]  e
[ 8: .......]  e
[ 9: .......]  n
------------
[10: .......]
[11: .......]  <- bottom history
[12: .......]
```

Note: Don't forget to update `Stream` class with appropriate escape sequences – you can use any, since pagination protocol is not standardized, for example:

```
Stream.escape["N"] = "next_page"
Stream.escape["P"] = "prev_page"
```

---

reset()
Overloaded to reset screen history state: history position is reset to bottom of both queues; queues themselves are emptied.

index()
Overloaded to update top history with the removed lines.

reverse_index()
Overloaded to update bottom history with the removed lines.

prev_page()
Moves the screen page up through the history buffer. Page size is defined by `history.ratio`, so for instance `ratio = .5` means that half the screen is restored from history on page switch.

next_page()
Moves the screen page down through the history buffer.

4.2.4 pyte.modes

This module defines terminal mode switches, used by `Screen`. There're two types of terminal modes:

- *non-private* which should be set with `ESC [ N h`, where `N` is an integer, representing mode being set; and
- *private* which should be set with `ESC [ ? N h`.

The latter are shifted 5 times to the right, to be easily distinguishable from the former ones; for example *Origin Mode* – `DECOM` is `192` not `6`.

```
>>> DECOM
192
```

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

`pyte.modes.`**LNM = 20**
Line Feed/New Line Mode: When enabled, causes a received `LF`, `pyte.control.FF`, or `VT` to move the cursor to the first column of the next line.

`pyte.modes.`**IRM = 4**
Insert/Replace Mode: When enabled, new display characters move old display characters to the right. Characters moved past the right margin are lost. Otherwise, new display characters replace old display characters at the cursor position.

`pyte.modes.`**DECTCEM = 800**
Text Cursor Enable Mode: determines if the text cursor is visible.

`pyte.modes.`**DECSCNM = 160**
Screen Mode: toggles screen-wide reverse-video mode.

`pyte.modes.`**DECOM = 192**
Origin Mode: allows cursor addressing relative to a user-defined origin. This mode resets when the terminal is powered up or reset. It does not affect the erase in display (ED) function.

`pyte.modes.`**DECAWM = 224**
Auto Wrap Mode: selects where received graphic characters appear when the cursor is at the right margin.

`pyte.modes.`**DECCOLM = 96**
Column Mode: selects the number of columns per line (80 or 132) on the screen.

4.2.5 pyte.control

This module defines simple control sequences, recognized by `Stream`, the set of codes here is for `TERM=linux` which is a superset of VT102.

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

`pyte.control.`**SP = u' '**
Space: Not suprisingly – `" "`.

`pyte.control.`**NUL = u'\x00'**
Null: Does nothing.

pyte.control.**BEL = u'\x07'**
Bell: Beeps.

pyte.control.**BS = u'\x08'**
Backspace: Backspace one column, but not past the begining of the line.

pyte.control.**HT = u'\t'**
Horizontal tab: Move cursor to the next tab stop, or to the end of the line if there is no earlier tab stop.

pyte.control.**LF = u'\n'**
Linefeed: Give a line feed, and, if `pyte.modes.LNM` (new line mode) is set also a carriage return.

pyte.control.**VT = u'\x0b'**
Vertical tab: Same as `LF`.

pyte.control.**FF = u'\x0c'**
Form feed: Same as `LF`.

pyte.control.**CR = u'\r'**
Carriage return: Move cursor to left margin on current line.

pyte.control.**SO = u'\x0e'**
Shift out: Activate G1 character set.

pyte.control.**SI = u'\x0f'**
Shift in: Activate G0 character set.

pyte.control.**CAN = u'\x18'**
Cancel: Interrupt escape sequence. If received during an escape or control sequence, cancels the sequence and displays substitution character.

pyte.control.**SUB = u'\x1a'**
Substitute: Same as `CAN`.

pyte.control.**ESC = u'\x1b'**
Escape: Starts an escape sequence.

pyte.control.**DEL = u'\x7f'**
Delete: Is ingored.

pyte.control.**CSI = u'\x9b'**
Control sequence introducer: An equavalent for `ESC [`.

4.2.6 pyte.escape

This module defines bot CSI and non-CSI escape sequences, recognized by `Stream` and subclasses.

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

pyte.escape.**RIS = u'c'**
Reset.

pyte.escape.**IND = u'D'**
Index: Move cursor down one line in same column. If the cursor is at the bottom margin, the screen performs a scroll-up.

pyte.escape.**NEL = u'E'**
Next line: Same as `pyte.control.LF`.

pyte.escape.**HTS = u'H'**
Tabulation set: Set a horizontal tab stop at cursor position.

pyte.escape.**RI = u'M'**
Reverse index: Move cursor up one line in same column. If the cursor is at the top margin, the screen performs a scroll-down.

pyte.escape.**DECSC = u'7'**
Save cursor: Save cursor position, character attribute (graphic rendition), character set, and origin mode selection (see DECRC).

pyte.escape.**DECRC = u'8'**
Restore cursor: Restore previously saved cursor position, character attribute (graphic rendition), character set, and origin mode selection. If none were saved, move cursor to home position.

pyte.escape.**DECALN = u'8'**
Alignment display: Fill screen with uppercase E's for testing screen focus and alignment.

pyte.escape.**ICH = u'@'**
Insert character: Insert the indicated # of blank characters.

pyte.escape.**CUU = u'A'**
Cursor up: Move cursor up the indicated # of lines in same column. Cursor stops at top margin.

pyte.escape.**CUD = u'B'**
Cursor down: Move cursor down the indicated # of lines in same column. Cursor stops at bottom margin.

pyte.escape.**CUF = u'C'**
Cursor forward: Move cursor right the indicated # of columns. Cursor stops at right margin.

pyte.escape.**CUB = u'D'**
Cursor back: Move cursor left the indicated # of columns. Cursor stops at left margin.

pyte.escape.**CNL = u'E'**
Cursor next line: Move cursor down the indicated # of lines to column 1.

pyte.escape.**CPL = u'F'**
Cursor previous line: Move cursor up the indicated # of lines to column 1.

pyte.escape.**CHA = u'G'**
Cursor horizontal align: Move cursor to the indicated column in current line.

pyte.escape.**CUP = u'H'**
Cursor position: Move cursor to the indicated line, column (origin at 1, 1).

pyte.escape.**ED = u'J'**
Erase data (default: from cursor to end of line).

pyte.escape.**EL = u'K'**
Erase in line (default: from cursor to end of line).

pyte.escape.**IL = u'L'**
Insert line: Insert the indicated # of blank lines, starting from the current line. Lines displayed below cursor move down. Lines moved past the bottom margin are lost.

pyte.escape.**DL = u'M'**
Delete line: Delete the indicated # of lines, starting from the current line. As lines are deleted, lines displayed below cursor move up. Lines added to bottom of screen have spaces with same character attributes as last line move up.

pyte.escape.**DCH = u'P'**
Delete character: Delete the indicated # of characters on the current line. When character is deleted, all characters to the right of cursor move left.

pyte.escape.**ECH = u'X'**
Erase character: Erase the indicated # of characters on the current line.

pyte.escape.**HPR = u'a'**
Horizontal position relative: Same as CUF.

pyte.escape.**VPA = u'd'**
Vertical position adjust: Move cursor to the indicated line, current column.

pyte.escape.**VPR = u'e'**
Vertical position relative: Same as CUD.

pyte.escape.**HVP = u'f'**
Horizontal / Vertical position: Same as CUP.

pyte.escape.**TBC = u'g'**
Tabulation clear: Clears a horizontal tab stop at cursor position.

pyte.escape.**SM = u'h'**
Set mode.

pyte.escape.**RM = u'l'**
Reset mode.

pyte.escape.**SGR = u'm'**
Select graphics rendition: The terminal can display the following character attributes that change the character display without changing the character (see pyte.graphics).

pyte.escape.**DECSTBM = u'r'**
Select top and bottom margins: Selects margins, defining the scrolling region; parameters are top and bottom line. If called without any arguments, whole screen is used.

pyte.escape.**HPA = u''''**
Horizontal position adjust: Same as CHA.

4.2.7 pyte.graphics

This module defines graphic-related constants, mostly taken from *console_codes(4)* and http://pueblo.sourceforge.net/doc/manual/ansi_color_codes.html.

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

pyte.graphics.**TEXT = {27: '-reverse', 1: '+bold', 3: '+italics', 4: '+underscore', 22: '-bold', 7: '+reverse', 24: '-underscore**
A mapping of ANSI text style codes to style names, "+" means the: attribute is set, "-" – reset; example:

```
>>> text[1]
'+bold'
>>> text[9]
'+strikethrough'
```

pyte.graphics.**FG = {32: 'green', 33: 'brown', 34: 'blue', 35: 'magenta', 36: 'cyan', 37: 'white', 39: 'default', 30: 'black', 3**
A mapping of ANSI foreground color codes to color names, example:

```
>>> FG[30]
'black'
>>> FG[38]
'default'
```

pyte.graphics.**BG = {49: 'default', 40: 'black', 41: 'red', 42: 'green', 43: 'brown', 44: 'blue', 45: 'magenta', 46: 'cyan', 47:**
A mapping of ANSI background color codes to color names, example:

```
>>> BG[40]
'black'
>>> BG[48]
'default'
```

4.2.8 pyte.charsets

This module defines `G0` and `G1` charset mappings the same way they are defined for linux terminal, see `linux/drivers/tty/consolemap.c` @ http://git.kernel.org

---

Note: `VT100_MAP` and `IBMPC_MAP` were taken unchanged from linux kernel source and therefore are licensed under **GPL**.

---

copyright

3. 2011-2013 by Selectel, see AUTHORS for details.

license LGPL, see LICENSE for more details.

pyte.charsets.**LAT1_MAP = [u'\x00', u'\x01', u'\x02', u'\x03', u'\x04', u'\x05', u'\x06', u'\x07', u'\x08', u'\t', u'\n', u'\x0b',**
Latin1.

pyte.charsets.**VT100_MAP = u'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x**
VT100 graphic character set.

pyte.charsets.**IBMPC_MAP = u'\x00\u263a\u263b\u2665\u2666\u2663\u2660\u2022\u25d8\u25cb\u25d9\u2642\u2640\u266**
IBM Codepage 437.

pyte.charsets.**VAX42_MAP = u'\x00\u263a\u263b\u2665\u2666\u2663\u2660\u2022\u25d8\u25cb\u25d9\u2642\u2640\u266**
VAX42 character set.

4.3 pyte Changelog

Here you can see the full list of changes between each pyte release.

4.3.1 Version 0.4.8

Released on January 13th 2014

- *pyte.screens.Screen* does NOT inherit from builtin *list*, use *screen.buffer* to access individual characters directly. This is a backward INCOMPATIBLE change.
- *pyte.screens.Char._asdict* was broken on Python 3.3 because of the changes in *namedtuple* implementation.
- *pyte.charsets.LAT1_MAP* was an iterator because of the change in map semantics in Python 3
- Changed *pyte.screens.Screen* to issues a CR in addition to LF when *DECAWM* mode is set and the cursor is at the right border of the screen. See http://www.vt100.net/docs/vt510-rm/DECAWM and issue #20 on GitHub for details.

4.3.2 Version 0.4.7

Bugfix release, released on March 28th 2013

- Updated *pyte* and tests suite to work under Python 3.3.
- Changed *pyte.screens.Screen* so that *LNM* mode is reset by default, see http://www.vt100.net/docs/vt510-rm/LNM and issue #11 on GitHub for details.

4.3.3 Version 0.4.6

Bugfix release, released on February 29th 2012

4.3.4 Version 0.4.5

Technical release, released on September 1st 2011

- Created MANIFEST.in file
- Added CentOS spec file

4.3.5 Version 0.4.4

Bugfix release, released on July 17th 2011

- Removed *pdb* calls, left from *HistoryScreen* debugging – silly, I know :)

4.3.6 Version 0.4.3

Bugfix release, released on July 12th 2011

- Fixed encoding issues in *DebugStream* – Unicode was not converted to bytes properly.
- Fixed G0-1 charset handling and added VAX42 charset for the ancient stuff to work correctly.

4.3.7 Version 0.4.2

Bugfix release, released on June 27th 2011

- Added a tiny debugging helper: `python -m pyte your escape codes`
- Added `Screen.__{before,after}__()` hooks to `Screen` – now subclasses can extend more than one command easily.
- Fixed `HistoryScreen` – now not as buggy as it used to be: and allows for custom ratio aspect when browsing history, see `HistoryScreen` documentation for details.
- Fixed *DECTCEM* private mode handling – when the mode is reset `Screen.cursor.hidden` is `True` otherwise it's `False`.

4.3.8 Version 0.4.1

Bugfix release, released on June 21st 2011

- Minor examples and documentation update before the first public release.

4.3.9 Version 0.4.0

Released on June 21st 2011

- Improved cursor movement – `Screen` passes all but one tests in *vttest*.
- Changed the way `Stream` interacts with `Screen` – event handlers are now implicitly looked up in screen's `__dict__`, not connected manually.
- Changed cursor API – cursor position and attributes are encapsulated in a separate `Cursor` class.
- Added support for *DECSCNM* – toggle screen-wide reverse-video mode.
- Added a couple of useful `Screen` subclasses: `HistoryScreen` which allows screen pagination and `DiffScreen` which tracks the changed lines.

4.3.10 Version 0.3.9

Released on May 31st 2011

- Added initial support for G0-1 charsets (mappings taken from `tty` kernel driver) and SI, SO escape sequences.
- Changed `ByteStream` to support fallback encodings – it nows takes a list of `(encoding, errors)` pairs and traverses it left to right on `feed()`.
- Switched to `unicode_literals` – one step closer to Python3.

4.3.11 Version 0.3.8

Released on May 23rd 2011

- Major rewrite of `Screen` internals – highlights: inherits from `list`; each character is represented by `namedtuple` which also holds SGR data.
- Numerous bugfixes, especialy in methods, dealing with manipulating character attributes.

4.3.12 Version 0.3.7

First release after the adoption – skipped a few version to reflect that. Released on May 16th 2011

- Added support for ANSI color codes, as listed in `man console_codes`. Not implemnted yet: setting alternate font, setting and resetting mappings, blinking text.
- Added a couple of trivial usage examples in the *examples/* dir.

p