tdl Documentation

Release 4.1.0

Kyle Stewart

Jul 20, 2017

Readme

Contents:

Contents

CHAPTER 1

Status

CHAPTER 2

About

This is a Python cffi port of libtcod.

This library is hosted on GitHub.

Any issues you have with this module can be reported at the GitHub issue tracker.

CHAPTER 3

Usage

This module was designed to be backward compatible with the original libtcodpy module distributed with libtcod. If you had code that runs on libtcodpy then you can use this library as a drop-in replacement:

```
import tcod as libtcod
```

Guides and Tutorials for libtcodpy should work with the tcod module.

The latest documentation can be found here.

CHAPTER 4

Installation

The recommended way to install is by using pip. Older versions of pip will have issues installing tdl, so make sure it's up-to-date.

Windows / MacOS

To install using pip, use the following command:

```
> python -m pip install tdl
```

Linux

On Linux, tdl will need to be built from source. Assuming you have Python and pip, you run these commands to install tdl:

```
$ sudo apt install gcc libsdl2-dev libffi-dev python-dev libomp-dev
$ pip install tdl
```

CHAPTER 5

Requirements

- Python 2.7+, Python 3.4+, or PyPy 5.4+
- Windows, Linux, or MacOS.
- Linux requires the libsdl2 package and must be installed from source.

CHAPTER 6

License

python-tdl is distributed under the Simplified 2-clause FreeBSD license.

CHAPTER 7

Changelog

4.1.0 - 2017-07-19

Changed

- Added protection to the *transparent*, *walkable*, and *fov* attributes in tcod and tdl Map classes, to prevent them from being accidentally overridden.
- tcod and tdl Map classes now use numpy arrays as their attributes.

4.0.1 - 2017-07-12

Fixed

- tdl: Fixed NameError in *set_fps*.

4.0.0 - 2017-07-08

Changed

- tcod.bsp: *BSP.split_recursive* parameter *random* is now *seed*.
- tcod.console: *Console.blit* parameters have been rearranged. Most of the parameters are now optional.
- tcod.noise: *Noise.__init__* parameter *rand* is now named *seed*.
- tdl: Changed *set_fps* paramter name to *fps*.

Fixed

- tcod.bsp: Corrected spelling of max_vertical_ratio.

3.2.0 - 2017-07-04

Changed

- Merged libtcod-cffi dependency with TDL.

Fixed

- Fixed boolean related crashes with Key 'text' events.
- tdl.noise: Fixed crash when given a negative seed. As well as cases where an instance could lose its seed being pickled.

3.1.0 - 2017-05-28

Added

- You can now pass tdl Console instances as parameters to libtcod-cffi functions expecting a tcod Console.

Changed

- Dependencies updated: *libtcod-cffi>=2.5.0,<3*
- The *Console.tcod_console* attribute is being renamed to *Console.console_c*.

Deprecated

- The tdl.noise and tdl.map modules will be deprecated in the future.

Fixed

- Resolved crash-on-exit issues for Windows platforms.

3.0.2 - 2017-04-13

Changed

- Dependencies updated: *libtcod-cffi>=2.4.3,<3*
- You can now create Console instances before a call to *tdl.init*.

Removed

- Dropped support for Python 3.3

Fixed

- Resolved issues with MacOS builds.
- 'OpenGL' and 'GLSL' renderers work again.

3.0.1 - 2017-03-22

Changed

- *KeyEvent*'s with *text* now have all their modifier keys set to False.

Fixed

- Undefined behaviour in text events caused crashes on 32-bit builds.

3.0.0 - 2017-03-21

Added

- *KeyEvent* supports libtcod text and meta keys.

Changed

- *KeyEvent* parameters have been moved.
- This version requires *libtcod-cffi>=2.3.0*.

Deprecated

- *KeyEvent* camel capped attribute names are deprecated.

Fixed

- Crashes with key-codes undefined by libtcod.
- *tdl.map* typedef issues with libtcod-cffi.

2.0.1 - 2017-02-22

Fixed

- *tdl.init* renderer was defaulted to OpenGL which is not supported in the current version of libtcod.

2.0.0 - 2017-02-15

Changed

- Dependencies updated, tdl now requires libtcod-cffi 2.x.x
- Some event behaviours have changed with SDL2, event keys might be different than what you expect.

Removed

- Key repeat functions were removed from SDL2. *set_key_repeat* is now stubbed, and does nothing.

1.6.0 - 2016-11-18

- Console.blit methods can now take fg_alpha and bg_alpha parameters.

1.5.3 - 2016-06-04

- set_font no longer crashes when loading a file without the implied font size in its name

1.5.2 - 2016-03-11

- Fixed non-square Map instances

1.5.1 - 2015-12-20

- Fixed errors with Unicode and non-Unicode literals on Python 2
- Fixed attribute error in compute_fov

1.5.0 - 2015-07-13

- python-tdl distributions are now universal builds
- New Map class
- map.bresenham now returns a list
- This release will require libtcod-cffi v0.2.3 or later

1.4.0 - 2015-06-22

- The DLL's have been moved into another library which you can find at https://github.com/HexDecimal/libtcod-cffi You can use this library to have some raw access to libtcod if you want. Plus it can be used alongside TDL.
- The libtocd console objects in Console instances have been made public.
- Added tdl.event.wait function. This function can called with a timeout and can automatically call tdl.flush.

1.3.1 - 2015-06-19

- Fixed pathfinding regressions.

1.3.0 - 2015-06-19

- Updated backend to use python-cffi instead of ctypes. This gives decent boost to speed in CPython and a drastic to boost in speed in PyPy.

1.2.0 - 2015-06-06

- The set_colors method now changes the default colors used by the draw_* methods. You can use Python's Ellipsis to explicitly select default colors this way.
- Functions and Methods renamed to match Python's style-guide PEP 8, the old function names still exist and are depreciated.

- The fgcolor and bgcolor parameters have been shortened to fg and bg.

1.1.7 - 2015-03-19

- Noise generator now seeds properly.
- The OS event queue will now be handled during a call to tdl.flush. This prevents a common newbie programmer hang where events are handled infrequently during long animations, simulations, or early development.
- Fixed a major bug that would cause a crash in later versions of Python 3

1.1.6 - 2014-06-27

- Fixed a race condition when importing on some platforms.
- Fixed a type issue with quickFOV on Linux.
- Added a bresenham function to the tdl.map module.

1.1.5 - 2013-11-10

- A for loop can iterate over all coordinates of a Console.
- drawStr can be configured to scroll or raise an error.
- You can now configure or disable key repeating with tdl.event.setKeyRepeat
- Typewriter class removed, use a Window instance for the same functionality.
- setColors method fixed.

1.1.4 - 2013-03-06

- Merged the Typewriter and MetaConsole classes, You now have a virtual cursor with Console and Window objects.
- Fixed the clear method on the Window class.
- Fixed screenshot function.
- Fixed some drawing operations with unchanging backgrounds.
- Instances of Console and Noise can be pickled and copied.
- Added KeyEvent.keychar
- Fixed event.keyWait, and now converts window closed events into Alt+F4.

1.1.3 - 2012-12-17

- Some of the setFont parameters were incorrectly labeled and documented.
- setFont can auto-detect tilesets if the font sizes are in the filenames.

- Added some X11 unicode tilesets, including unifont.

1.1.2 - 2012-12-13

- Window title now defaults to the running scripts filename.
- Fixed incorrect deltaTime for App.update
- App will no longer call tdl.flush on its own, you'll need to call this yourself.
- tdl.noise module added.
- clear method now defaults to black on black.

1.1.1 - 2012-12-05

- Map submodule added with AStar class and quickFOV function.
- New Typewriter class.
- Most console functions can use Python-style negative indexes now.
- New App.runOnce method.
- Rectangle geometry is less strict.

1.1.0 - 2012-10-04

- KeyEvent.keyname is now KeyEvent.key
- MouseButtonEvent.button now behaves like KeyEvent.keyname does.
- event.App class added.
- Drawing methods no longer have a default for the character parameter.
- KeyEvent.ctrl is now KeyEvent.control

1.0.8 - 2010-04-07

- No longer works in Python 2.5 but now works in 3.x and has been partly tested.
- Many bug fixes.

1.0.5 - 2010-04-06

- Got rid of setuptools dependency, this will make it much more compatible with Python 3.x
- Fixed a typo with the MacOS library import.

1.0.4 - 2010-04-06

- All constant colors (C_*) have been removed, they may be put back in later.
- Made some type assertion failures show the value they received to help in general debugging. Still working on it.
- Added MacOS and 64-bit Linux support.

1.0.0 - 2009-01-31

- First public release.

CHAPTER 8

Glossary

color control

color controls

`tcod.`**`COLCTRL_1`**

`tcod.`**`COLCTRL_2`**

`tcod.`**`COLCTRL_3`**

`tcod.`**`COLCTRL_4`**

`tcod.`**`COLCTRL_5`**
: Configurable color control constant which can be set up with *`tcod.console_set_color_control`*.

`tcod.`**`COLCTRL_STOP`**

`tcod.`**`COLCTRL_FORE_RGB`**

`tcod.`**`COLCTRL_BACK_RGB`**

console defaults The default values implied by any Console print or put functions which don't explicitly ask for them as parameters.

CHAPTER 9

tcod.console

libtcod works with a special 'root' console. You create this console using the *tcod.console_init_root* function. Usually after setting the font with *console_set_custom_font* first.

Example:

```
# Make sure 'arial10x10.png' is in the same directory as this script.
import time

import tcod

# Setup the font.
tcod.console_set_custom_font(
    'arial10x10.png',
    tcod.FONT_LAYOUT_ASCII_INROW | tcod.FONT_LAYOUT_TCOD,
    )
# Initialize the root console in a context.
with tcod.console_init_root(80, 60, 'title') as root_console:
    root_console.print_(x=0, y=0, string='Hello World!')
    tcod.console_flush() # Show the console.
    time.sleep(3) # Wait 3 seconds.
# The window is closed here, after the above context exits.
```

class `tcod.console.`**`Console`**(*width*, *height*)

Parameters

- **width** (*int*) – Width of the new Console.
- **height** (*int*) – Height of the new Console.

`console_c`
CData – A cffi pointer to a TCOD_console_t object.

`__bool__`()
Returns False if this is the root console.

This mimics libtcodpy behavior.

__enter__()
Returns this console in a managed context.

When the root console is used as a context, the graphical window will close once the context is left as if *tcod.console_delete* was called on it.

This is useful for some Python IDE's like IDLE, where the window would not be closed on its own otherwise.

__exit__(**args*)
Closes the graphical window on exit.

Some tcod functions may have undefined behavior after this point.

__nonzero__()
Returns False if this is the root console.

This mimics libtcodpy behavior.

blit(*dest*, *dest_x=0*, *dest_y=0*, *src_x=0*, *src_y=0*, *width=0*, *height=0*, *fg_alpha=1.0*, *bg_alpha=1.0*, *key_color=None*)
Blit from this console onto the `dest` console.

Parameters

- **dest** (*Console*) – The destintaion console to blit onto.
- **dest_x** (*int*) – Leftmost coordinate of the destintaion console.
- **dest_y** (*int*) – Topmost coordinate of the destintaion console.
- **src_x** (*int*) – X coordinate from this console to blit, from the left.
- **src_y** (*int*) – Y coordinate from this console to blit, from the top.
- **width** (*int*) – The width of the region to blit.

  If this is 0 the maximum possible width will be used.
- **height** (*int*) – The height of the region to blit.

  If this is 0 the maximum possible height will be used.
- **fg_alpha** (*float*) – Foreground color alpha vaule.
- **bg_alpha** (*float*) – Background color alpha vaule.
- **key_color** (*Optional[Tuple[int, int, int]]*) – None, or a (red, green, blue) tuple with values of 0-255.

Changed in version 4.0: Parameters were rearraged and made optional.

Previously they were: *(x, y, width, height, dest, dest_x, dest_y, *)*

clear()
Reset this console to its default colors and the space character.

get_height_rect(*x*, *y*, *width*, *height*, *string*)
Return the height of this text word-wrapped into this rectangle.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **width** (*int*) – Maximum width to render the text.
- **height** (*int*) – Maximum lines to render the text.

- **string** (*Text*) – A Unicode string.

Returns The number of lines of text once word-wrapped.

Return type int

hline(*x*, *y*, *width*, *bg_blend=13*)
Draw a horizontal line on the console.

This always uses the character 196, the horizontal line character.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **width** (*int*) – The horozontal length of this line.
- **bg_blend** (*int*) – The background blending flag.

print_(*x*, *y*, *string*, *bg_blend=13*, *alignment=None*)
Print a color formatted string on a console.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **string** (*Text*) – A Unicode string optionaly using color codes.

print_frame(*x*, *y*, *width*, *height*, *string=''*, *clear=True*, *bg_blend=13*)
Draw a framed rectangle with optinal text.

This uses the default background color and blend mode to fill the rectangle and the default foreground to draw the outline.

string will be printed on the inside of the rectangle, word-wrapped.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **width** (*int*) – The width if the frame.
- **height** (*int*) – The height of the frame.
- **string** (*Text*) – A Unicode string to print.
- **clear** (*bool*) – If True all text in the affected area will be removed.
- **bg_blend** (*int*) – The background blending flag.

Note: This method does not support Unicode outside of the 0-255 range.

---

print_rect(*x*, *y*, *width*, *height*, *string*, *bg_blend=13*, *alignment=None*)
Print a string constrained to a rectangle.

If h > 0 and the bottom of the rectangle is reached, the string is truncated. If h = 0, the string is only truncated if it reaches the bottom of the console.

Parameters

- **x** (*int*) – The x coordinate from the left.

- **y** (*int*) – The y coordinate from the top.
- **width** (*int*) – Maximum width to render the text.
- **height** (*int*) – Maximum lines to render the text.
- **string** (*Text*) – A Unicode string.
- **bg_blend** (*int*) – Background blending flag.
- **alignment** (*Optional[int]*) – Alignment flag.

Returns The number of lines of text once word-wrapped.

Return type int

put_char(*x*, *y*, *ch*, *bg_blend=13*)

Draw the character c at x,y using the default colors and a blend mode.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **ch** (*int*) – Character code to draw. Must be in integer form.
- **bg_blend** (*int*) – Blending mode to use, defaults to BKGND_DEFAULT.

rect(*x*, *y*, *width*, *height*, *clear*, *bg_blend=13*)

Draw a the background color on a rect optionally clearing the text.

If clr is True the affected tiles are changed to space character.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **width** (*int*) – Maximum width to render the text.
- **height** (*int*) – Maximum lines to render the text.
- **clear** (*bool*) – If True all text in the affected area will be removed.
- **bg_blend** (*int*) – Background blending flag.

set_key_color(*color*)

Set a consoles blit transparent color.

Parameters **color** (*Tuple[int, int, int]*) –

vline(*x*, *y*, *height*, *bg_blend=13*)

Draw a vertical line on the console.

This always uses the character 179, the vertical line character.

Parameters

- **x** (*int*) – The x coordinate from the left.
- **y** (*int*) – The y coordinate from the top.
- **height** (*int*) – The horozontal length of this line.
- **bg_blend** (*int*) – The background blending flag.

bg
A uint8 array with the shape (height, width, 3).

You can change the consoles background colors by using this array.

Index this array with `console.bg[y, x, channel]`

ch
An integer array with the shape (height, width).

You can change the consoles character codes by using this array.

Index this array with `console.ch[y, x]`

default_alignment
int – The default text alignment.

default_bg
Tuple[int, int, int] – The default background color.

default_bg_blend
int – The default blending mode.

default_fg
Tuple[int, int, int] – The default foreground color.

fg
A uint8 array with the shape (height, width, 3).

You can change the consoles foreground colors by using this array.

Index this array with `console.fg[y, x, channel]`

height
int – The height of this Console. (read-only)

width
int – The width of this Console. (read-only)

CHAPTER 10

tcod.map

libtcod map attributes and field-of-view functions.

Example:

```
>>> import tcod.map
>>> m = tcod.map.Map(width=3, height=4)
>>> m.walkable
array([[False, False, False],
       [False, False, False],
       [False, False, False],
       [False, False, False]], dtype=bool)

# Like the rest of the tcod modules, all arrays here are
# in row-major order and are addressed with [y,x]
>>> m.transparent[:] = True # Sets all to True.
>>> m.transparent[1:3,0] = False # Sets (1, 0) and (2, 0) to False.
>>> m.transparent
array([[ True,  True,  True],
       [False,  True,  True],
       [False,  True,  True],
       [ True,  True,  True]], dtype=bool)

>>> m.compute_fov(0, 0)
>>> m.fov
array([[ True,  True,  True],
       [ True,  True,  True],
       [False,  True,  True],
       [False, False,  True]], dtype=bool)
>>> m.fov[3,1]
False
```

class tcod.map.**Map**(*width*, *height*)

A map containing libtcod attributes.

Changed in version 4.1: *transparent*, *walkable*, and *fov* are now numpy boolean arrays.

Parameters

- **width** (*int*) – Width of the new Map.
- **height** (*int*) – Height of the new Map.

width
int – Read only width of this Map.

height
int – Read only height of this Map.

transparent
A boolean array of transparent cells.

walkable
A boolean array of walkable cells.

fov
A boolean array of the cells lit by :any:'compute_fov'.

compute_fov(*x*, *y*, *radius=0*, *light_walls=True*, *algorithm=12*)
Compute a field-of-view on the current instance.

Parameters

- **x** (*int*) – Point of view, x-coordinate.
- **y** (*int*) – Point of view, y-coordinate.
- **radius** (*int*) – Maximum view distance from the point of view.

  A value of *0* will give an infinite distance.
- **light_walls** (*bool*) – Light up walls, or only the floor.
- **algorithm** (*int*) – Defaults to tcod.FOV_RESTRICTIVE

CHAPTER 11

tcod.bsp

The following example shows how to travrse the BSP tree using Python. This assumes *create_room* and *connect_rooms* will be replaced by custom code.

Example:

```
import tcod.bsp

def create_room(node):
    """Initialize the room at this current node."""
    print('Create a room for %s.' % node)

def connect_rooms(node):
    """Connect two fully initialized rooms."""
    node1, node2 = node.children
    print('Connect the rooms:\n%s\n%s' % (node1, node2))

def traverse(node):
    """Traverse a BSP tree dispatching nodes to the correct calls."""
    # For nodes without children, node.children is an empty tuple.
    for child in node.children:
        traverse(child)

    if node.children:
        connect_rooms(node)
    else:
        create_room(node)

bsp = tcod.bsp.BSP(x=0, y=0, width=80, height=60)
bsp.split_recursive(
    depth=5,
    min_width=3,
    min_height=3,
    max_horizontal_ratio=1.5,
    max_vertical_ratio=1.5,
    )
```

```
traverse(bsp)
```

class tcod.bsp.**BSP**(*x*, *y*, *width*, *height*)

A binary space partitioning tree which can be used for simple dungeon generation.

x
int – Rectangle left coordinate.

y
int – Rectangle top coordinate.

width
int – Rectangle width.

height
int – Rectangle height.

level
int – This nodes depth.

position
int – The integer of where the node was split.

horizontal
bool – This nodes split orientation.

parent
Optional[BSP] – This nodes parent or None

children
Optional[Tuple[BSP, BSP]] – A tuple of (left, right) BSP instances, or None if this BSP has no children.

Parameters

- **x** (*int*) – Rectangle left coordinate.
- **y** (*int*) – Rectangle top coordinate.
- **width** (*int*) – Rectangle width.
- **height** (*int*) – Rectangle height.

__str__()
Provide a useful readout when printed.

contains(*x*, *y*)
Returns True if this node contains these coordinates.

Parameters

- **x** (*int*) – X position to check.
- **y** (*int*) – Y position to check.

Returns

True if this node contains these coordinates. Otherwise False.

Return type bool

find_node(*x*, *y*)
Return the deepest node which contains these coordinates.

Returns BSP object or None.

Return type Optional[*BSP*]

split_once(*horizontal*, *position*)
Split this partition into 2 sub-partitions.

Parameters

- **horizontal** (*bool*) –
- **position** (*int*) –

split_recursive(*depth*, *min_width*, *min_height*, *max_horizontal_ratio*, *max_vertical_ratio*, *seed=None*)
Divide this partition recursively.

Parameters

- **depth** (*int*) – The maximum depth to divide this object recursively.
- **min_width** (*int*) – The minimum width of any individual partition.
- **min_height** (*int*) – The minimum height of any individual partition.
- **max_horizontal_ratio** (*float*) – Prevent creating a horizontal ratio more extreme than this.
- **max_vertical_ratio** (*float*) – Prevent creating a vertical ratio more extreme than this.
- **seed** (*Optional[tcod.random.Random]*) – The random number generator to use.

walk()
Iterate over this BSP's hieracrhy.

The iterator will include the instance which called it. It will traverse its own children and grandchildren, in no particular order.

Returns An iterator of BSP nodes.

Return type Iterator[*BSP*]

Deprecated since version 2.3: See the module example for how to iterate over a BSP tree.

CHAPTER 12

tcod.path

Example:

```
>>> import numpy as np
>>> import tcod.path
>>> dungeon = np.array(
...     [
...         [1, 0, 1, 1, 1],
...         [1, 0, 1, 0, 1],
...         [1, 1, 1, 0, 1],
...     ],
...     dtype=np.int8,
...     )
...
>>> height, width = dungeon.shape

# Create a pathfinder from a numpy array.
# This is the recommended way to use the tcod.path module.
>>> astar = tcod.path.AStar(dungeon)
>>> print(astar.get_path(0, 0, 4, 2))
[(0, 1), (1, 2), (2, 1), (3, 0), (4, 1), (4, 2)]
>>> astar.cost[0, 1] = 1 # You can access the map array via this attribute.
>>> print(astar.get_path(0, 0, 4, 2))
[(1, 0), (2, 0), (3, 0), (4, 1), (4, 2)]

# Create a pathfinder from an edge_cost function.
# Calling Python functions from C is known to be very slow.
>>> def edge_cost(my_x, my_y, dest_x, dest_y):
...     return dungeon[dest_y, dest_x]
...
>>> dijkstra = tcod.path.Dijkstra(
...     tcod.path.EdgeCostCallback(edge_cost, width, height),
...     )
...
>>> dijkstra.set_goal(0, 0)
>>> print(dijkstra.get_path(4, 2))
```

```
[(1, 0), (2, 0), (3, 0), (4, 1), (4, 2)]
```

class tcod.path.**AStar**(*cost*, *diagonal=1.41*)

Parameters

- **cost** (*Union[tcod.map.Map, numpy.ndarray, Any]*) –
- **diagonal** (*float*) – Multiplier for diagonal movement. A value of 0 will disable diagonal movement entirely.

get_path(*start_x*, *start_y*, *goal_x*, *goal_y*)
Return a list of (x, y) steps to reach the goal point, if possible.

Parameters

- **start_x** (*int*) – Starting X position.
- **start_y** (*int*) – Starting Y position.
- **goal_x** (*int*) – Destination X position.
- **goal_y** (*int*) – Destination Y position.

Returns A list of points, or an empty list if there is no valid path.

Return type List[Tuple[int, int]]

class tcod.path.**Dijkstra**(*cost*, *diagonal=1.41*)

Parameters

- **cost** (*Union[tcod.map.Map, numpy.ndarray, Any]*) –
- **diagonal** (*float*) – Multiplier for diagonal movement. A value of 0 will disable diagonal movement entirely.

get_path(*x*, *y*)
Return a list of (x, y) steps to reach the goal point, if possible.

set_goal(*x*, *y*)
Set the goal point and recompute the Dijkstra path-finder.

class tcod.path.**EdgeCostCallback**(*callback*, *width*, *height*)
Calculate cost from an edge-cost callback.

Parameters

- **callback** (*Callable[[int, int, int, int], float]*) – A callback which can handle the following parameters: *(source_x:int, source_y:int, dest_x:int, dest_y:int) -> float*
- **width** (*int*) – The maximum width of this callback.
- **height** (*int*) – The maximum height of this callback.

class tcod.path.**NodeCostArray**
Calculate cost from a numpy array of nodes.

Parameters **array** (*numpy.ndarray*) – A numpy array with the cost of each node.

CHAPTER 13

tcod.image

class tcod.image.**Image**(*width*, *height*)

Parameters

- **width** (*int*) – Width of the new Image.
- **height** (*int*) – Height of the new Image.

width
int – Read only width of this Image.

height
int – Read only height of this Image.

blit(*console*, *x*, *y*, *bg_blend*, *scale_x*, *scale_y*, *angle*)
Blit onto a Console using scaling and rotation.

Parameters

- **console** (Console) – Blit destination Console.
- **x** (*int*) – Console X position for the center of the Image blit.
- **y** (*int*) – Console Y position for the center of the Image blit. The Image blit is centered on this position.
- **bg_blend** (*int*) – Background blending mode to use.
- **scale_x** (*float*) – Scaling along Image x axis. Set to 1 for no scaling. Must be over 0.
- **scale_y** (*float*) – Scaling along Image y axis. Set to 1 for no scaling. Must be over 0.
- **angle** (*float*) – Rotation angle in radians. (Clockwise?)

blit_2x(*console*, *dest_x*, *dest_y*, *img_x=0*, *img_y=0*, *img_width=-1*, *img_height=-1*)
Blit onto a Console with double resolution.

Parameters

- **console** (Console) – Blit destination Console.

- **dest_x** (*int*) – Console tile X position starting from the left at 0.
- **dest_y** (*int*) – Console tile Y position starting from the top at 0.
- **img_x** (*int*) – Left corner pixel of the Image to blit
- **img_y** (*int*) – Top corner pixel of the Image to blit
- **img_width** (*int*) – Width of the Image to blit. Use -1 for the full Image width.
- **img_height** (*int*) – Height of the Image to blit. Use -1 for the full Image height.

blit_rect(*console*, *x*, *y*, *width*, *height*, *bg_blend*)
Blit onto a Console without scaling or rotation.

Parameters

- **console** (*Console*) – Blit destination Console.
- **x** (*int*) – Console tile X position starting from the left at 0.
- **y** (*int*) – Console tile Y position starting from the top at 0.
- **width** (*int*) – Use -1 for Image width.
- **height** (*int*) – Use -1 for Image height.
- **bg_blend** (*int*) – Background blending mode to use.

clear(*color*)
Fill this entire Image with color.

Parameters **color** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

get_alpha(*x*, *y*)
Get the Image alpha of the pixel at x, y.

Parameters

- **x** (*int*) – X pixel of the image. Starting from the left at 0.
- **y** (*int*) – Y pixel of the image. Starting from the top at 0.

Returns The alpha value of the pixel. With 0 being fully transparent and 255 being fully opaque.

Return type int

get_mipmap_pixel(*left*, *top*, *right*, *bottom*)
Get the average color of a rectangle in this Image.

Parameters should stay within the following limits: * 0 <= left < right < Image.width * 0 <= top < bottom < Image.height

Parameters

- **left** (*int*) – Left corner of the region.
- **top** (*int*) – Top corner of the region.
- **right** (*int*) – Right corner of the region.
- **bottom** (*int*) – Bottom corner of the region.

Returns An (r, g, b) tuple containing the averaged color value. Values are in a 0 to 255 range.

Return type Tuple[int, int, int]

get_pixel(*x*, *y*)
Get the color of a pixel in this Image.

Parameters

- **x** (*int*) – X pixel of the Image. Starting from the left at 0.
- **y** (*int*) – Y pixel of the Image. Starting from the top at 0.

Returns An (r, g, b) tuple containing the pixels color value. Values are in a 0 to 255 range.

Return type Tuple[int, int, int]

hflip()
Horizontally flip this Image.

invert()
Invert all colors in this Image.

put_pixel(*x*, *y*, *color*)
Change a pixel on this Image.

Parameters

- **x** (*int*) – X pixel of the Image. Starting from the left at 0.
- **y** (*int*) – Y pixel of the Image. Starting from the top at 0.
- **color** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

refresh_console(*console*)
Update an Image created with *tcod.image_from_console*.

The console used with this function should have the same width and height as the Console given to *tcod.image_from_console*. The font width and height must also be the same as when *tcod.image_from_console* was called.

Parameters **console** (*Console*) – A Console with a pixel width and height matching this Image.

rotate90(*rotations=1*)
Rotate this Image clockwise in 90 degree steps.

Parameters **rotations** (*int*) – Number of 90 degree clockwise rotations.

save_as(*filename*)
Save the Image to a 32-bit .bmp or .png file.

Parameters **filename** (*Text*) – File path to same this Image.

scale(*width*, *height*)
Scale this Image to the new width and height.

Parameters

- **width** (*int*) – The new width of the Image after scaling.
- **height** (*int*) – The new height of the Image after scaling.

set_key_color(*color*)
Set a color to be transparent during blitting functions.

Parameters **color** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

vflip()
Vertically flip this Image.

CHAPTER 14

tcod.random

Random module docs.

class tcod.random.**Random**(*algorithm*, *seed=None*)

The libtcod random number generator.

If all you need is a random number generator then it's recommended that you use the *random* module from the Python standard library.

If `seed` is None then a random seed will be generated.

Parameters

- **algorithm** (*int*) – The algorithm to use.
- **seed** (*Optional[Hashable]*) – Could be a 32-bit integer, but any hashable object is accepted.

random_c

CData – A cffi pointer to a TCOD_random_t object.

__getstate__()

Pack the self.random_c attribute into a portable state.

__setstate__(*state*)

Create a new cdata object with the stored paramaters.

guass(*mu*, *sigma*)

Return a random number using Gaussian distribution.

Parameters

- **mu** (*float*) – The median returned value.
- **sigma** (*float*) – The standard deviation.

Returns A random float.

Return type float

inverse_guass(*mu*, *sigma*)
Return a random Gaussian number using the Box-Muller transform.

Parameters

- **mu** (*float*) – The median returned value.
- **sigma** (*float*) – The standard deviation.

Returns A random float.

Return type float

randint(*low*, *high*)
Return a random integer within the linear range: low <= n <= high.

Parameters

- **low** (*int*) – The lower bound of the random range.
- **high** (*int*) – The upper bound of the random range.

Returns A random integer.

Return type int

uniform(*low*, *high*)
Return a random floating number in the range: low <= n <= high.

Parameters

- **low** (*int*) – The lower bound of the random range.
- **high** (*int*) – The upper bound of the random range.

Returns A random float.

Return type float

CHAPTER 15

tcod.noise

The *Noise.sample_mgrid* and *Noise.sample_ogrid* methods are multi-threaded operations when the Python runtime supports OpenMP. Even when single threaded these methods will perform much better than multiple calls to *Noise.get_point*.

Example:

```
import numpy as np
import tcod
import tcod.noise

noise = tcod.noise.Noise(
    dimensions=2,
    algorithm=tcod.NOISE_SIMPLEX,
    implementation=tcod.noise.TURBULENCE,
    hurst=0.5,
    lacunarity=2.0,
    octaves=4,
    seed=None,
    )

# Create a 5x5 open multi-dimensional mesh-grid.
ogrid = [np.arange(5, dtype=np.float32),
         np.arange(5, dtype=np.float32)]
print(ogrid)

# Scale the grid.
ogrid[0] *= 0.25
ogrid[1] *= 0.25

# Return the sampled noise from this grid of points.
samples = noise.sample_ogrid(ogrid)
print(samples)
```

class tcod.noise.**Noise**(*dimensions*, *algorithm=2*, *implementation=0*, *hurst=0.5*, *lacunarity=2.0*, *octaves=4*, *seed=None*)

The `hurst` exponent describes the raggedness of the resultant noise, with a higher value leading to a smoother noise. Not used with tcod.noise.SIMPLE.

`lacunarity` is a multiplier that determines how fast the noise frequency increases for each successive octave. Not used with tcod.noise.SIMPLE.

Parameters

- **dimensions** (*int*) – Must be from 1 to 4.
- **algorithm** (*int*) – Defaults to NOISE_SIMPLEX
- **implementation** (*int*) – Defaults to tcod.noise.SIMPLE
- **hurst** (*float*) – The hurst exponent. Should be in the 0.0-1.0 range.
- **lacunarity** (*float*) – The noise lacunarity.
- **octaves** (*float*) – The level of detail on fBm and turbulence implementations.
- **seed** (*Optional[Random]*) – A Random instance, or None.

noise_c
CData – A cffi pointer to a TCOD_noise_t object.

get_point(*x=0*, *y=0*, *z=0*, *w=0*)
Return the noise value at the (x, y, z, w) point.

Parameters

- **x** (*float*) – The position on the 1st axis.
- **y** (*float*) – The position on the 2nd axis.
- **z** (*float*) – The position on the 3rd axis.
- **w** (*float*) – The position on the 4th axis.

sample_mgrid(*mgrid*)
Sample a mesh-grid array and return the result.

The *sample_ogrid* method performs better as there is a lot of overhead when working with large mesh-grids.

Parameters **mgrid** (*numpy.ndarray*) – A mesh-grid array of points to sample. A contiguous array of type *numpy.float32* is preferred.

Returns

An array of sampled points.

This array has the shape: `mgrid.shape[:-1]`. The `dtype` is *numpy.float32*.

Return type numpy.ndarray

sample_ogrid(*ogrid*)
Sample an open mesh-grid array and return the result.

Args ogrid (Sequence[Sequence[float]]): An open mesh-grid.

Returns

An array of sampled points.

The `shape` is based on the lengths of the open mesh-grid arrays. The `dtype` is *numpy.float32*.

Return type numpy.ndarray

CHAPTER 16

libtcodpy

bsp

tcod.**bsp_new_with_size**(*x*, *y*, *w*, *h*)
Create a new BSP instance with the given rectangle.

Parameters

- **x** (*int*) – Rectangle left coordinate.
- **y** (*int*) – Rectangle top coordinate.
- **w** (*int*) – Rectangle width.
- **h** (*int*) – Rectangle height.

Returns A new BSP instance.

Return type *BSP*

Deprecated since version 2.0: Call the *BSP* class instead.

tcod.**bsp_split_once**(*node*, *horizontal*, *position*)
Deprecated since version 2.0: Use *BSP.split_once* instead.

tcod.**bsp_split_recursive**(*node*, *randomizer*, *nb*, *minHSize*, *minVSize*, *maxHRatio*, *maxVRatio*)
Deprecated since version 2.0: Use *BSP.split_recursive* instead.

tcod.**bsp_resize**(*node*, *x*, *y*, *w*, *h*)
Deprecated since version 2.0: Assign directly to *BSP* attributes instead.

tcod.**bsp_left**(*node*)
Deprecated since version 2.0: Use *BSP.children* instead.

tcod.**bsp_right**(*node*)
Deprecated since version 2.0: Use *BSP.children* instead.

tcod.**bsp_father**(*node*)
Deprecated since version 2.0: Use *BSP.parent* instead.

tcod.**bsp_is_leaf**(*node*)
Deprecated since version 2.0: Use *BSP.children* instead.

tcod.**bsp_contains**(*node*, *cx*, *cy*)
Deprecated since version 2.0: Use *BSP.contains* instead.

tcod.**bsp_find_node**(*node*, *cx*, *cy*)
Deprecated since version 2.0: Use *BSP.find_node* instead.

tcod.**bsp_traverse_pre_order**(*node*, *callback*, *userData=0*)
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use *BSP.walk* instead.

tcod.**bsp_traverse_in_order**(*node*, *callback*, *userData=0*)
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use *BSP.walk* instead.

tcod.**bsp_traverse_post_order**(*node*, *callback*, *userData=0*)
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use *BSP.walk* instead.

tcod.**bsp_traverse_level_order**(*node*, *callback*, *userData=0*)
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use *BSP.walk* instead.

tcod.**bsp_traverse_inverted_level_order**(*node*, *callback*, *userData=0*)
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use *BSP.walk* instead.

tcod.**bsp_remove_sons**(*node*)
Delete all children of a given node. Not recommended.

Note: This function will add unnecessary complexity to your code. Don't use it.

Deprecated since version 2.0: BSP deletion is automatic.

tcod.**bsp_delete**(*node*)
Exists for backward compatibility. Does nothing.

BSP's created by this library are automatically garbage collected once there are no references to the tree. This function exists for backwards compatibility.

Deprecated since version 2.0: BSP deletion is automatic.

color

class tcod.**Color**(*r=0*, *g=0*, *b=0*)

Parameters

- **r** (*int*) – Red value, from 0 to 255.
- **g** (*int*) – Green value, from 0 to 255.
- **b** (*int*) – Blue value, from 0 to 255.

r
int – Red value, always normalised to 0-255.

g
int – Green value, always normalised to 0-255.

b
int – Blue value, always normalised to 0-255.

__eq__(*other*)
Compare equality between colors.

Also compares with standard sequences such as 3-item tuples or lists.

__add__(*other*)
Add two colors together.

__sub__(*other*)
Subtract one color from another.

__mul__(*other*)
Multiply with a scaler or another color.

__repr__()
Return a printable representation of the current color.

tcod.**color_lerp**(*c1*, *c2*, *a*)
Return the linear interpolation between two colors.

`a` is the interpolation value, with 0 returing `c1`, 1 returning `c2`, and 0.5 returing a color halfway between both.

Parameters

- **c1** (*Union[Tuple[int, int, int], Sequence[int]]*) – The first color. At a=0.
- **c2** (*Union[Tuple[int, int, int], Sequence[int]]*) – The second color. At a=1.
- **a** (*float*) – The interpolation value,

Returns The interpolated Color.

Return type *Color*

tcod.**color_set_hsv**(*c*, *h*, *s*, *v*)
Set a color using: hue, saturation, and value parameters.

Does not return a new Color. `c` is modified inplace.

Parameters

- **c** (*Union[Color, List[Any]]*) – A Color instance, or a list of any kind.
- **h** (*float*) – Hue, from 0 to 360.
- **s** (*float*) – Saturation, from 0 to 1.
- **v** (*float*) – Value, from 0 to 1.

tcod.**color_get_hsv**(*c*)
Return the (hue, saturation, value) of a color.

Parameters **c** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

Returns A tuple with (hue, saturation, value) values, from 0 to 1.

Return type Tuple[float, float, float]

tcod.**color_scale_HSV**(*c*, *scoef*, *vcoef*)
Scale a color's saturation and value.

Does not return a new Color. `c` is modified inplace.

Parameters

- **c** (*Union[Color, List[int]]*) – A Color instance, or an [r, g, b] list.
- **scoef** (*float*) – Saturation multiplier, from 0 to 1. Use 1 to keep current saturation.
- **vcoef** (*float*) – Value multiplier, from 0 to 1. Use 1 to keep current value.

tcod.**color_gen_map**(*colors*, *indexes*)
Return a smoothly defined scale of colors.

If `indexes` is [0, 3, 9] for example, the first color from `colors` will be returned at 0, the 2nd will be at 3, and the 3rd will be at 9. All in-betweens will be filled with a gradient.

Parameters

- **colors** (*Iterable[Union[Tuple[int, int, int], Sequence[int]]]*) – Array of colors to be sampled.
- **indexes** (*Iterable[int]*) – A list of indexes.

Returns A list of Color instances.

Return type List[*Color*]

Example

```
>>> tcod.color_gen_map([(0, 0, 0), (255, 128, 0)], [0, 5])
[Color(0,0,0), Color(51,25,0), Color(102,51,0), Color(153,76,0), Color(204,102,0),
↪ Color(255,128,0)]
```

console

tcod.**console_set_custom_font**(*fontFile*, *flags=1*, *nb_char_horiz=0*, *nb_char_vertic=0*)
Load a custom font file.

Call this before function before calling *tcod.console_init_root*.

Flags can be a mix of the following:

•tcod.FONT_LAYOUT_ASCII_INCOL

•tcod.FONT_LAYOUT_ASCII_INROW

•tcod.FONT_TYPE_GREYSCALE

•tcod.FONT_TYPE_GRAYSCALE

•tcod.FONT_LAYOUT_TCOD

Parameters

- **fontFile** (*AnyStr*) – Path to a font file.
- **flags** (*int*) –

- **nb_char_horiz** (*int*) –
- **nb_char_vertic** (*int*) –

tcod.**console_init_root**(*w*, *h*, *title*, *fullscreen=False*, *renderer=0*)
Set up the primary display and return the root console.

Parameters

- **w** (*int*) – Width in character tiles for the root console.
- **h** (*int*) – Height in character tiles for the root console.
- **title** (*AnyStr*) – This string will be displayed on the created windows title bar.
- **renderer** – Rendering mode for libtcod to use.

Returns Returns a special Console instance representing the root console.

Return type *Console*

tcod.**console_flush**()
Update the display to represent the root consoles current state.

tcod.**console_blit**(*src*, *x*, *y*, *w*, *h*, *dst*, *xdst*, *ydst*, *ffade=1.0*, *bfade=1.0*)
Blit the console src from x,y,w,h to console dst at xdst,ydst.

tcod.**console_check_for_keypress**(*flags=2*)

tcod.**console_clear**(*con*)
Reset a console to its default colors and the space character.

Parameters **con** (Console) – Any Console instance.

See also:

console_set_default_background *console_set_default_foreground*

tcod.**console_credits**()

tcod.**console_credits_render**(*x*, *y*, *alpha*)

tcod.**console_credits_reset**()

tcod.**console_delete**(*con*)

tcod.**console_fill_background**(*con*, *r*, *g*, *b*)
Fill the backgound of a console with r,g,b.

Parameters

- **con** (Console) – Any Console instance.
- **r** (*Sequence[int]*) – An array of integers with a length of width*height.
- **g** (*Sequence[int]*) – An array of integers with a length of width*height.
- **b** (*Sequence[int]*) – An array of integers with a length of width*height.

tcod.**console_fill_char**(*con*, *arr*)
Fill the character tiles of a console with an array.

Parameters

- **con** (Console) – Any Console instance.
- **arr** (*Sequence[int]*) – An array of integers with a length of width*height.

tcod.**console_fill_foreground**(*con*, *r*, *g*, *b*)
Fill the foregound of a console with r,g,b.

Parameters

- **con** (Console) – Any Console instance.
- **r** (*Sequence[int]*) – An array of integers with a length of width*height.
- **g** (*Sequence[int]*) – An array of integers with a length of width*height.
- **b** (*Sequence[int]*) – An array of integers with a length of width*height.

tcod.**console_from_file**(*filename*)
Return a new console object from a filename.

The file format is automactially determined. This can load REXPaint *.xp*, ASCII Paint *.apf*, or Non-delimited ASCII *.asc* files.

Parameters **filename** (*Text*) – The path to the file, as a string.

Returns: A new :any‘Console‘ instance.

tcod.**console_from_xp**(*filename*)
Return a single console from a REXPaint *.xp* file.

tcod.**console_get_alignment**(*con*)
Return this consoles current alignment mode.

Parameters **con** (Console) – Any Console instance.

tcod.**console_get_background_flag**(*con*)
Return this consoles current blend mode.

Parameters **con** (Console) – Any Console instance.

tcod.**console_get_char**(*con*, *x*, *y*)
Return the character at the x,y of this console.

tcod.**console_get_char_background**(*con*, *x*, *y*)
Return the background color at the x,y of this console.

tcod.**console_get_char_foreground**(*con*, *x*, *y*)
Return the foreground color at the x,y of this console.

tcod.**console_get_default_background**(*con*)
Return this consoles default background color.

tcod.**console_get_default_foreground**(*con*)
Return this consoles default foreground color.

tcod.**console_get_fade**()

tcod.**console_get_fading_color**()

tcod.**console_get_height**(*con*)
Return the height of a console.

Parameters **con** (Console) – Any Console instance.

Returns The height of a Console.

Return type int

Deprecated since version 2.0: Use *Console.get_hright* instead.

tcod.**console_get_height_rect**(*con*, *x*, *y*, *w*, *h*, *fmt*)
Return the height of this text once word-wrapped into this rectangle.

Returns The number of lines of text once word-wrapped.

Return type int

tcod.**console_get_width**(*con*)
Return the width of a console.

Parameters **con** (Console) – Any Console instance.

Returns The width of a Console.

Return type int

Deprecated since version 2.0: Use *Console.get_width* instead.

tcod.**console_hline**(*con*, *x*, *y*, *l*, *flag=13*)
Draw a horizontal line on the console.

This always uses the character 196, the horizontal line character.

tcod.**console_is_fullscreen**()
Returns True if the display is fullscreen.

Returns True if the display is fullscreen, otherwise False.

Return type bool

tcod.**console_is_key_pressed**(*key*)

tcod.**console_is_window_closed**()
Returns True if the window has received and exit event.

tcod.**console_load_apf**(*con*, *filename*)
Update a console from an ASCII Paint *.apf* file.

tcod.**console_load_asc**(*con*, *filename*)
Update a console from a non-delimited ASCII *.asc* file.

tcod.**console_load_xp**(*con*, *filename*)
Update a console from a REXPaint *.xp* file.

tcod.**console_list_load_xp**(*filename*)
Return a list of consoles from a REXPaint *.xp* file.

tcod.**console_list_save_xp**(*console_list*, *filename*, *compress_level=9*)
Save a list of consoles to a REXPaint *.xp* file.

tcod.**console_map_ascii_code_to_font**(*asciiCode*, *fontCharX*, *fontCharY*)
Set a character code to new coordinates on the tile-set.

asciiCode must be within the bounds created during the initialization of the loaded tile-set. For example, you can't use 255 here unless you have a 256 tile tile-set loaded. This applies to all functions in this group.

Parameters

- **asciiCode** (*int*) – The character code to change.
- **fontCharX** (*int*) – The X tile coordinate on the loaded tileset. 0 is the leftmost tile.
- **fontCharY** (*int*) – The Y tile coordinate on the loaded tileset. 0 is the topmost tile.

tcod.**console_map_ascii_codes_to_font**(*firstAsciiCode*, *nbCodes*, *fontCharX*, *fontCharY*)
Remap a contiguous set of codes to a contiguous set of tiles.

Both the tile-set and character codes must be contiguous to use this function. If this is not the case you may want to use *console_map_ascii_code_to_font*.

Parameters

- **firstAsciiCode** (*int*) – The starting character code.
- **nbCodes** (*int*) – The length of the contiguous set.
- **fontCharX** (*int*) – The starting X tile coordinate on the loaded tileset. 0 is the leftmost tile.
- **fontCharY** (*int*) – The starting Y tile coordinate on the loaded tileset. 0 is the topmost tile.

tcod.**console_map_string_to_font**(*s*, *fontCharX*, *fontCharY*)

Remap a string of codes to a contiguous set of tiles.

Parameters

- **s** (*AnyStr*) – A string of character codes to map to new values. The null character '' will prematurely end this function.
- **fontCharX** (*int*) – The starting X tile coordinate on the loaded tileset. 0 is the leftmost tile.
- **fontCharY** (*int*) – The starting Y tile coordinate on the loaded tileset. 0 is the topmost tile.

tcod.**console_new**(*w*, *h*)

Return an offscreen console of size: w,h.

tcod.**console_print**(*con*, *x*, *y*, *fmt*)

Print a color formatted string on a console.

Parameters

- **con** (Console) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.
- **fmt** (*AnyStr*) – A unicode or bytes string optionaly using color codes.

tcod.**console_print_ex**(*con*, *x*, *y*, *flag*, *alignment*, *fmt*)

Print a string on a console using a blend mode and alignment mode.

Parameters

- **con** (Console) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.

tcod.**console_print_frame**(*con*, *x*, *y*, *w*, *h*, *clear=True*, *flag=13*, *fmt=''*)

Draw a framed rectangle with optinal text.

This uses the default background color and blend mode to fill the rectangle and the default foreground to draw the outline.

fmt will be printed on the inside of the rectangle, word-wrapped.

tcod.**console_print_rect**(*con*, *x*, *y*, *w*, *h*, *fmt*)

Print a string constrained to a rectangle.

If h > 0 and the bottom of the rectangle is reached, the string is truncated. If h = 0, the string is only truncated if it reaches the bottom of the console.

Returns The number of lines of text once word-wrapped.

Return type int

tcod.**console_print_rect_ex**(*con*, *x*, *y*, *w*, *h*, *flag*, *alignment*, *fmt*)
Print a string constrained to a rectangle with blend and alignment.

Returns The number of lines of text once word-wrapped.

Return type int

tcod.**console_put_char**(*con*, *x*, *y*, *c*, *flag=13*)
Draw the character c at x,y using the default colors and a blend mode.

Parameters

- **con** (*Console*) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.
- **c** (*Union[int, AnyStr]*) – Character to draw, can be an integer or string.
- **flag** (*int*) – Blending mode to use, defaults to BKGND_DEFAULT.

tcod.**console_put_char_ex**(*con*, *x*, *y*, *c*, *fore*, *back*)
Draw the character c at x,y using the colors fore and back.

Parameters

- **con** (*Console*) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.
- **c** (*Union[int, AnyStr]*) – Character to draw, can be an integer or string.
- **fore** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.
- **back** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

tcod.**console_rect**(*con*, *x*, *y*, *w*, *h*, *clr*, *flag=13*)
Draw a the background color on a rect optionally clearing the text.

If clr is True the affected tiles are changed to space character.

tcod.**console_save_apf**(*con*, *filename*)
Save a console to an ASCII Paint *.apf* file.

tcod.**console_save_asc**(*con*, *filename*)
Save a console to a non-delimited ASCII *.asc* file.

tcod.**console_save_xp**(*con*, *filename*, *compress_level=9*)
Save a console to a REXPaint *.xp* file.

tcod.**console_set_alignment**(*con*, *alignment*)
Change this consoles current alignment mode.

•tcod.LEFT

•tcod.CENTER

•tcod.RIGHT

Parameters

- **con** (Console) – Any Console instance.
- **alignment** (*int*) –

tcod.**console_set_background_flag**(*con*, *flag*)
Change the default blend mode for this console.

Parameters

- **con** (Console) – Any Console instance.
- **flag** (*int*) – Blend mode to use by default.

tcod.**console_set_char**(*con*, *x*, *y*, *c*)
Change the character at x,y to c, keeping the current colors.

Parameters

- **con** (Console) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.
- **c** (*Union[int, AnyStr]*) – Character to draw, can be an integer or string.

tcod.**console_set_char_background**(*con*, *x*, *y*, *col*, *flag=1*)
Change the background color of x,y to col using a blend mode.

Parameters

- **con** (Console) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.
- **col** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.
- **flag** (*int*) – Blending mode to use, defaults to BKGND_SET.

tcod.**console_set_char_foreground**(*con*, *x*, *y*, *col*)
Change the foreground color of x,y to col.

Parameters

- **con** (Console) – Any Console instance.
- **x** (*int*) – Character x position from the left.
- **y** (*int*) – Character y position from the top.
- **col** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

tcod.**console_set_color_control**(*con*, *fore*, *back*)
Configure *color controls*.

Parameters

- **con** (*int*) – *Color control* constant to modify.

- **fore** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.
- **back** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

tcod.**console_set_default_background**(*con*, *col*)
Change the default background color for a console.

Parameters

- **con** (Console) – Any Console instance.
- **col** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

tcod.**console_set_default_foreground**(*con*, *col*)
Change the default foreground color for a console.

Parameters

- **con** (Console) – Any Console instance.
- **col** (*Union[Tuple[int, int, int], Sequence[int]]*) – An (r, g, b) sequence or Color instance.

tcod.**console_set_fade**(*fade*, *fadingColor*)

tcod.**console_set_fullscreen**(*fullscreen*)
Change the display to be fullscreen or windowed.

Parameters **fullscreen** (*bool*) – Use True to change to fullscreen. Use False to change to windowed.

tcod.**console_set_key_color**(*con*, *col*)
Set a consoles blit transparent color.

tcod.**console_set_window_title**(*title*)
Change the current title bar string.

Parameters **title** (*AnyStr*) – A string to change the title bar to.

tcod.**console_vline**(*con*, *x*, *y*, *l*, *flag=13*)
Draw a vertical line on the console.

This always uses the character 179, the vertical line character.

tcod.**console_wait_for_keypress**(*flush*)
Block until the user presses a key, then returns a new Key.

Parameters **bool** (flush) – If True then the event queue is cleared before waiting for the next event.

Returns A new Key instance.

Return type *Key*

Event

class tcod.**Key**
Key Event instance

vk
int – TCOD_keycode_t key code

c
int – character if vk == TCODK_CHAR else 0

text
Text – text[TCOD_KEY_TEXT_SIZE]; text if vk == TCODK_TEXT else text[0] == ‘’

pressed
bool – does this correspond to a key press or key release event ?

lalt
bool – True when left alt is held.

lctrl
bool – True when left control is held.

lmeta
bool – True when left meta key is held.

ralt
bool – True when right alt is held.

rctrl
bool – True when right control is held.

rmeta
bool – True when right meta key is held.

shift
bool – True when any shift is held.

__repr__()
Return a representation of this Key object.

class tcod.**Mouse**
Mouse event instance

x
int – Absolute mouse position at pixel x.

y
int

dx
int – Movement since last update in pixels.

dy
int

cx
int – Cell coordinates in the root console.

cy
int

dcx
int – Movement since last update in console cells.

dcy
int

lbutton
bool – Left button status.

rbutton
bool – Right button status.

mbutton
bool – Middle button status.

lbutton_pressed
bool – Left button pressed event.

rbutton_pressed
bool – Right button pressed event.

mbutton_pressed
bool – Middle button pressed event.

wheel_up
bool – Wheel up event.

wheel_down
bool – Wheel down event.

__repr__()
Return a representation of this Mouse object.

Event Types

tcod.**EVENT_NONE**

tcod.**EVENT_KEY_PRESS**

tcod.**EVENT_KEY_RELEASE**

tcod.**EVENT_KEY**
Same as `tcod.EVENT_KEY_PRESS | tcod.EVENT_KEY_RELEASE`

tcod.**EVENT_MOUSE_MOVE**

tcod.**EVENT_MOUSE_PRESS**

tcod.**EVENT_MOUSE_RELEASE**

tcod.**EVENT_MOUSE**
Same as `tcod.EVENT_MOUSE_MOVE | tcod.EVENT_MOUSE_PRESS | tcod.EVENT_MOUSE_RELEASE`

tcod.**EVENT_FINGER_MOVE**

tcod.**EVENT_FINGER_PRESS**

tcod.**EVENT_FINGER_RELEASE**

tcod.**EVENT_FINGER**
Same as `tcod.EVENT_FINGER_MOVE | tcod.EVENT_FINGER_PRESS | tcod.EVENT_FINGER_RELEASE`

tcod.**EVENT_ANY**
Same as `tcod.EVENT_KEY | tcod.EVENT_MOUSE | tcod.EVENT_FINGER`

sys

tcod.**sys_set_fps**(*fps*)
Set the maximum frame rate.

You can disable the frame limit again by setting fps to 0.

Parameters **fps** (*int*) – A frame rate limit (i.e. 60)

tcod.**sys_get_fps**()
Return the current frames per second.

This the actual frame rate, not the frame limit set by *tcod.sys_set_fps*.

This number is updated every second.

Returns The currently measured frame rate.

Return type int

tcod.**sys_get_last_frame_length**()
Return the delta time of the last rendered frame in seconds.

Returns The delta time of the last rendered frame.

Return type float

tcod.**sys_sleep_milli**(*val*)
Sleep for 'val' milliseconds.

Parameters **val** (*int*) – Time to sleep for in milliseconds.

Deprecated since version 2.0: Use `time.sleep` instead.

tcod.**sys_elapsed_milli**()
Get number of milliseconds since the start of the program.

Returns Time since the progeam has started in milliseconds.

Return type int

Deprecated since version 2.0: Use `time.clock` instead.

tcod.**sys_elapsed_seconds**()
Get number of seconds since the start of the program.

Returns Time since the progeam has started in seconds.

Return type float

Deprecated since version 2.0: Use `time.clock` instead.

tcod.**sys_set_renderer**(*renderer*)
Change the current rendering mode to renderer.

Deprecated since version 2.0: RENDERER_GLSL and RENDERER_OPENGL are not currently available.

tcod.**sys_get_renderer**()
Return the current rendering mode.

tcod.**sys_save_screenshot**(*name=None*)
Save a screenshot to a file.

By default this will automatically save screenshots in the working directory.

The automatic names are formatted as screenshotNNN.png. For example: screenshot000.png, screenshot001.png, etc. Whichever is available first.

Parameters **Optional[AnyStr]** (*file*) – File path to save screenshot.

tcod.**sys_force_fullscreen_resolution**(*width*, *height*)
Force a specific resolution in fullscreen.

Will use the smallest available resolution so that:

- resolution width >= width and resolution width >= root console width * font char width
- resolution height >= height and resolution height >= root console height * font char height

Parameters

- **width** (*int*) – The desired resolution width.
- **height** (*int*) – The desired resolution height.

tcod.**sys_get_current_resolution**()
Return the current resolution as (width, height)

Returns The current resolution.

Return type Tuple[int,int]

tcod.**sys_get_char_size**()
Return the current fonts character size as (width, height)

Returns The current font glyph size in (width, height)

Return type Tuple[int,int]

tcod.**sys_update_char**(*asciiCode*, *fontx*, *fonty*, *img*, *x*, *y*)
Dynamically update the current frot with img.

All cells using this asciiCode will be updated at the next call to *tcod.console_flush*.

Parameters

- **asciiCode** (*int*) – Ascii code corresponding to the character to update.
- **fontx** (*int*) – Left coordinate of the character in the bitmap font (in tiles)
- **fonty** (*int*) – Top coordinate of the character in the bitmap font (in tiles)
- **img** (*Image*) – An image containing the new character bitmap.
- **x** (*int*) – Left pixel of the character in the image.
- **y** (*int*) – Top pixel of the character in the image.

tcod.**sys_register_SDL_renderer**(*callback*)
Register a custom randering function with libtcod.

Note: This callback will only be called by the SDL renderer.

The callack will receive a CData void* to an SDL_Surface* struct.

The callback is called on every call to *tcod.console_flush*.

Parameters **Callable[[CData], None]** (*callback*) – A function which takes a single argument.

tcod.**sys_check_for_event**(*mask*, *k*, *m*)
Check for and return an event.

Parameters

- **mask** (*int*) – *Event Types* to wait for.
- **k** (*Optional[Key]*) – A tcod.Key instance which might be updated with an event. Can be None.
- **m** (*Optional[Mouse]*) – A tcod.Mouse instance which might be updated with an event. Can be None.

tcod.**sys_wait_for_event**(*mask*, *k*, *m*, *flush*)
Wait for an event then return.

If flush is True then the buffer will be cleared before waiting. Otherwise each available event will be returned in the order they're recieved.

Parameters

- **mask** (*int*) – *Event Types* to wait for.
- **k** (*Optional[Key]*) – A tcod.Key instance which might be updated with an event. Can be None.
- **m** (*Optional[Mouse]*) – A tcod.Mouse instance which might be updated with an event. Can be None.
- **flush** (*bool*) – Clear the event buffer before waiting.

pathfinding

tcod.**dijkstra_compute**(*p*, *ox*, *oy*)

tcod.**dijkstra_delete**(*p*)

tcod.**dijkstra_get**(*p*, *idx*)

tcod.**dijkstra_get_distance**(*p*, *x*, *y*)

tcod.**dijkstra_is_empty**(*p*)

tcod.**dijkstra_new**(*m*, *dcost=1.41*)

tcod.**dijkstra_new_using_function**(*w*, *h*, *func*, *userData=0*, *dcost=1.41*)

tcod.**dijkstra_path_set**(*p*, *x*, *y*)

tcod.**dijkstra_path_walk**(*p*)

tcod.**dijkstra_reverse**(*p*)

tcod.**dijkstra_size**(*p*)

tcod.**path_compute**(*p*, *ox*, *oy*, *dx*, *dy*)
Find a path from (ox, oy) to (dx, dy). Return True if path is found.

Parameters

- **p** (*AStar*) – An AStar instance.
- **ox** (*int*) – Starting x position.
- **oy** (*int*) – Starting y position.
- **dx** (*int*) – Destination x position.
- **dy** (*int*) – Destination y position.

Returns True if a valid path was found. Otherwise False.

Return type bool

tcod.**path_delete**(*p*)
Does nothing.

tcod.**path_get**(*p*, *idx*)
Get a point on a path.

Parameters

- **p** (*AStar*) – An AStar instance.
- **idx** (*int*) – Should be in range: 0 <= inx < *path_size*

tcod.**path_get_destination**(*p*)
Get the current destination position.

Parameters **p** (*AStar*) – An AStar instance.

Returns An (x, y) point.

Return type Tuple[int, int]

tcod.**path_get_origin**(*p*)
Get the current origin position.

This point moves when *path_walk* returns the next x,y step.

Parameters **p** (*AStar*) – An AStar instance.

Returns An (x, y) point.

Return type Tuple[int, int]

tcod.**path_is_empty**(*p*)
Return True if a path is empty.

Parameters **p** (*AStar*) – An AStar instance.

Returns True if a path is empty. Otherwise False.

Return type bool

tcod.**path_new_using_function**(*w*, *h*, *func*, *userData=0*, *dcost=1.41*)
Return a new AStar using the given callable function.

Parameters

- **w** (*int*) – Clipping width.
- **h** (*int*) – Clipping height.
- **func** (*Callable[[int, int, int, int, Any], float]*) –
- **userData** (*Any*) –
- **dcost** (*float*) – A multiplier for the cost of diagonal movement. Can be set to 0 to disable diagonal movement.

Returns A new AStar instance.

Return type *AStar*

tcod.**path_new_using_map**(*m*, *dcost=1.41*)
Return a new AStar using the given Map.

Parameters

- **m** (Map) – A Map instance.
- **dcost** (*float*) – The path-finding cost of diagonal movement. Can be set to 0 to disable diagonal movement.

Returns A new AStar instance.

Return type *AStar*

tcod.**path_reverse**(*p*)
Reverse the direction of a path.

This effectively swaps the origin and destination points.

Parameters **p** (AStar) – An AStar instance.

tcod.**path_size**(*p*)
Return the current length of the computed path.

Parameters **p** (AStar) – An AStar instance.

Returns Length of the path.

Return type int

tcod.**path_walk**(*p*, *recompute*)
Return the next (x, y) point in a path, or (None, None) if it's empty.

When `recompute` is True and a previously valid path reaches a point where it is now blocked, a new path will automatically be found.

Parameters

- **p** (AStar) – An AStar instance.
- **recompute** (*bool*) – Recompute the path automatically.

Returns A single (x, y) point, or (None, None)

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

heightmap

tcod.**heightmap_add**(*hm*, *value*)
Add value to all values on this heightmap.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **value** (*float*) – A number to add to this heightmap.

Deprecated since version 2.0: Do `hm[:] += value` instead.

tcod.**heightmap_add_fbm**(*hm*, *noise*, *mulx*, *muly*, *addx*, *addy*, *octaves*, *delta*, *scale*)
Add FBM noise to the heightmap.

The noise coordinate for each map cell is *((x + addx) * mulx / width, (y + addy) * muly / height)*.

The value added to the heightmap is *delta + noise * scale*.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **noise** (Noise) – A Noise instance.

- **mulx** (*float*) – Scaling of each x coordinate.
- **muly** (*float*) – Scaling of each y coordinate.
- **addx** (*float*) – Translation of each x coordinate.
- **addy** (*float*) – Translation of each y coordinate.
- **octaves** (*float*) – Number of octaves in the FBM sum.
- **delta** (*float*) – The value added to all heightmap cells.
- **scale** (*float*) – The noise value is scaled with this parameter.

tcod.**heightmap_add_hill**(*hm*, *x*, *y*, *radius*, *height*)
Add a hill (a half spheroid) at given position.

If height == radius or -radius, the hill is a half-sphere.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **x** (*float*) – The x position at the center of the new hill.
- **y** (*float*) – The y position at the center of the new hill.
- **radius** (*float*) – The size of the new hill.
- **height** (*float*) – The height or depth of the new hill.

tcod.**heightmap_add_hm**(*hm1*, *hm2*, *hm3*)
Add two heightmaps together and stores the result in `hm3`.

Parameters

- **hm1** (*numpy.ndarray*) – The first heightmap.
- **hm2** (*numpy.ndarray*) – The second heightmap to add to the first.
- **hm3** (*numpy.ndarray*) – A destination heightmap to store the result.

Deprecated since version 2.0: Do `hm3[:] = hm1[:] + hm2[:]` instead.

tcod.**heightmap_add_voronoi**(*hm*, *nbPoints*, *nbCoef*, *coef*, *rnd=None*)
Add values from a Voronoi diagram to the heightmap.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **nbPoints** (*Any*) – Number of Voronoi sites.
- **nbCoef** (*int*) – The diagram value is calculated from the nbCoef closest sites.
- **coef** (*Sequence[float]*) – The distance to each site is scaled by the corresponding coef. Closest site : coef[0], second closest site : coef[1], ...
- **rnd** (*Optional[Random]*) – A Random instance, or None.

tcod.**heightmap_clamp**(*hm*, *mi*, *ma*)
Clamp all values on this heightmap between `mi` and `ma`

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **mi** (*float*) – The lower bound to clamp to.
- **ma** (*float*) – The upper bound to clamp to.

Deprecated since version 2.0: Do `hm.clip(mi, ma)` instead.

`tcod.`**`heightmap_clear`**(*hm*)
Add value to all values on this heightmap.

Parameters **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.

Deprecated since version 2.0: Do `hm.array[:] = 0` instead.

`tcod.`**`heightmap_copy`**(*hm1*, *hm2*)
Copy the heightmap `hm1` to `hm2`.

Parameters

- **hm1** (*numpy.ndarray*) – The source heightmap.
- **hm2** (*numpy.ndarray*) – The destination heightmap.

Deprecated since version 2.0: Do `hm2[:] = hm1[:]` instead.

`tcod.`**`heightmap_count_cells`**(*hm*, *mi*, *ma*)
Return the number of map cells which value is between `mi` and `ma`.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **mi** (*float*) – The lower bound.
- **ma** (*float*) – The upper bound.

Returns The count of values which fall between `mi` and `ma`.

Return type int

`tcod.`**`heightmap_delete`**(*hm*)
Does nothing.

Deprecated since version 2.0: libtcod-cffi deletes heightmaps automatically.

`tcod.`**`heightmap_dig_bezier`**(*hm*, *px*, *py*, *startRadius*, *startDepth*, *endRadius*, *endDepth*)
Carve a path along a cubic Bezier curve.

Both radius and depth can vary linearly along the path.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **px** (*Sequence[int]*) – The 4 *x* coordinates of the Bezier curve.
- **py** (*Sequence[int]*) – The 4 *y* coordinates of the Bezier curve.
- **startRadius** (*float*) – The starting radius size.
- **startDepth** (*float*) – The starting depth.
- **endRadius** (*float*) – The ending radius size.
- **endDepth** (*float*) – The ending depth.

`tcod.`**`heightmap_dig_hill`**(*hm*, *x*, *y*, *radius*, *height*)
This function takes the highest value (if height > 0) or the lowest (if height < 0) between the map and the hill.

It's main goal is to carve things in maps (like rivers) by digging hills along a curve.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.

- **x** (*float*) – The x position at the center of the new carving.
- **y** (*float*) – The y position at the center of the new carving.
- **radius** (*float*) – The size of the carving.
- **height** (*float*) – The height or depth of the hill to dig out.

tcod.**heightmap_get_interpolated_value**(*hm*, *x*, *y*)
Return the interpolated height at non integer coordinates.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **x** (*float*) – A floating point x coordinate.
- **y** (*float*) – A floating point y coordinate.

Returns The value at `x`, `y`.

Return type float

tcod.**heightmap_get_minmax**(*hm*)
Return the min and max values of this heightmap.

Parameters **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.

Returns The (min, max) values.

Return type Tuple[float, float]

Deprecated since version 2.0: Do `hm.min()` or `hm.max()` instead.

tcod.**heightmap_get_normal**(*hm*, *x*, *y*, *waterLevel*)
Return the map normal at given coordinates.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **x** (*float*) – The x coordinate.
- **y** (*float*) – The y coordinate.
- **waterLevel** (*float*) – The heightmap is considered flat below this value.

Returns An (x, y, z) vector normal.

Return type Tuple[float, float, float]

tcod.**heightmap_get_slope**(*hm*, *x*, *y*)
Return the slope between 0 and (pi / 2) at given coordinates.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **x** (*int*) – The x coordinate.
- **y** (*int*) – The y coordinate.

Returns The steepness at `x`, `y`. From 0 to (pi / 2)

Return type float

tcod.**heightmap_get_value**(*hm*, *x*, *y*)
Return the value at `x`, `y` in a heightmap.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **x** (*int*) – The x position to pick.
- **y** (*int*) – The y position to pick.

Returns The value at `x`, `y`.

Return type float

Deprecated since version 2.0: Do `value = hm[y, x]` instead.

`tcod.`**`heightmap_has_land_on_border`**(*hm*, *waterlevel*)

Returns True if the map edges are below `waterlevel`, otherwise False.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **waterLevel** (*float*) – The water level to use.

Returns True if the map edges are below `waterlevel`, otherwise False.

Return type bool

`tcod.`**`heightmap_kernel_transform`**(*hm*, *kernelsize*, *dx*, *dy*, *weight*, *minLevel*, *maxLevel*)

Apply a generic transformation on the map, so that each resulting cell value is the weighted sum of several neighbour cells.

This can be used to smooth/sharpen the map.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **kernelsize** (*int*) –

  Should be set to the length of the parameters:: dx, dy, and weight.
- **dx** (*Sequence[int]*) – A sequence of x coorinates.
- **dy** (*Sequence[int]*) – A sequence of y coorinates.
- **weight** (*Sequence[float]*) – A sequence of kernelSize cells weight. The value of each neighbour cell is scaled by its corresponding weight
- **minLevel** (*float*) – No transformation will apply to cells below this value.
- **maxLevel** (*float*) – No transformation will apply to cells above this value.

See examples below for a simple horizontal smoothing kernel : replace value(x,y) with 0.33*value(x-1,y) + 0.33*value(x,y) + 0.33*value(x+1,y). To do this, you need a kernel of size 3 (the sum involves 3 surrounding cells). The dx,dy array will contain * dx=-1, dy=0 for cell (x-1, y) * dx=1, dy=0 for cell (x+1, y) * dx=0, dy=0 for cell (x, y) * The weight array will contain 0.33 for each cell.

Example

```
>>> import numpy as np
>>> heightmap = np.zeros((3, 3), dtype=np.float32)
>>> heightmap[:,1] = 1
>>> dx = [-1, 1, 0]
>>> dy = [0, 0, 0]
>>> weight = [0.33, 0.33, 0.33]
>>> tcod.heightmap_kernel_transform(heightmap, 3, dx, dy, weight,
...                                 0.0, 1.0)
```

tcod.**heightmap_lerp_hm**(*hm1*, *hm2*, *hm3*, *coef*)
Perform linear interpolation between two heightmaps storing the result in `hm3`.

This is the same as doing `hm3[:] = hm1[:] + (hm2[:] - hm1[:]) * coef`

Parameters

- **hm1** (*numpy.ndarray*) – The first heightmap.
- **hm2** (*numpy.ndarray*) – The second heightmap to add to the first.
- **hm3** (*numpy.ndarray*) – A destination heightmap to store the result.
- **coef** (*float*) – The linear interpolation coefficient.

tcod.**heightmap_multiply_hm**(*hm1*, *hm2*, *hm3*)
Multiplies two heightmap's together and stores the result in `hm3`.

Parameters

- **hm1** (*numpy.ndarray*) – The first heightmap.
- **hm2** (*numpy.ndarray*) – The second heightmap to multiply with the first.
- **hm3** (*numpy.ndarray*) – A destination heightmap to store the result.

Deprecated since version 2.0: Do `hm3[:] = hm1[:] * hm2[:]` instead. Alternatively you can do `HeightMap(hm1.array[:] * hm2.array[:])`.

tcod.**heightmap_new**(*w*, *h*)
Return a new numpy.ndarray formatted for use with heightmap functions.

You can pass a numpy array to any heightmap function as long as all the following are true:: * The array is 2 dimentional. * The array has the C_CONTIGUOUS flag. * The array's dtype is `dtype.float32`.

Parameters

- **w** (*int*) – The width of the new HeightMap.
- **h** (*int*) – The height of the new HeightMap.

Returns A C-contiguous mapping of float32 values.

Return type numpy.ndarray

tcod.**heightmap_normalize**(*hm*, *mi=0.0*, *ma=1.0*)
Normalize heightmap values between `mi` and `ma`.

Parameters

- **mi** (*float*) – The lowest value after normalization.
- **ma** (*float*) – The highest value after normalization.

tcod.**heightmap_rain_erosion**(*hm*, *nbDrops*, *erosionCoef*, *sedimentationCoef*, *rnd=None*)
Simulate the effect of rain drops on the terrain, resulting in erosion.

`nbDrops` should be at least hm.size.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **nbDrops** (*int*) – Number of rain drops to simulate.
- **erosionCoef** (*float*) – Amount of ground eroded on the drop's path.

- **sedimentationCoef** (*float*) – Amount of ground deposited when the drops stops to flow.
- **rnd** (*Optional[Random]*) – A tcod.Random instance, or None.

tcod.**heightmap_scale**(*hm*, *value*)
Multiply all items on this heightmap by value.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **value** (*float*) – A number to scale this heightmap by.

Deprecated since version 2.0: Do `hm[:] *= value` instead.

tcod.**heightmap_scale_fbm**(*hm*, *noise*, *mulx*, *muly*, *addx*, *addy*, *octaves*, *delta*, *scale*)
Multiply the heighmap values with FBM noise.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **noise** (Noise) – A Noise instance.
- **mulx** (*float*) – Scaling of each x coordinate.
- **muly** (*float*) – Scaling of each y coordinate.
- **addx** (*float*) – Translation of each x coordinate.
- **addy** (*float*) – Translation of each y coordinate.
- **octaves** (*float*) – Number of octaves in the FBM sum.
- **delta** (*float*) – The value added to all heightmap cells.
- **scale** (*float*) – The noise value is scaled with this parameter.

tcod.**heightmap_set_value**(*hm*, *x*, *y*, *value*)
Set the value of a point on a heightmap.

Parameters

- **hm** (*numpy.ndarray*) – A numpy.ndarray formatted for heightmap functions.
- **x** (*int*) – The x position to change.
- **y** (*int*) – The y position to change.
- **value** (*float*) – The value to set.

Deprecated since version 2.0: Do `hm[y, x] = value` instead.

image

tcod.**image_load**(*filename*)
Load an image file into an Image instance and return it.

Parameters **filename** (*AnyStr*) – Path to a .bmp or .png image file.

tcod.**image_from_console**(*console*)
Return an Image with a Consoles pixel data.

This effectively takes a screen-shot of the Console.

Parameters **console** (Console) – Any Console instance.

tcod.**image_blit**(*image*, *console*, *x*, *y*, *bkgnd_flag*, *scalex*, *scaley*, *angle*)

tcod.**image_blit_2x**(*image*, *console*, *dx*, *dy*, *sx=0*, *sy=0*, *w=-1*, *h=-1*)

tcod.**image_blit_rect**(*image*, *console*, *x*, *y*, *w*, *h*, *bkgnd_flag*)

tcod.**image_clear**(*image*, *col*)

tcod.**image_delete**(*image*)

tcod.**image_get_alpha**(*image*, *x*, *y*)

tcod.**image_get_mipmap_pixel**(*image*, *x0*, *y0*, *x1*, *y1*)

tcod.**image_get_pixel**(*image*, *x*, *y*)

tcod.**image_get_size**(*image*)

tcod.**image_hflip**(*image*)

tcod.**image_invert**(*image*)

tcod.**image_is_pixel_transparent**(*image*, *x*, *y*)

tcod.**image_new**(*width*, *height*)

tcod.**image_put_pixel**(*image*, *x*, *y*, *col*)

tcod.**image_refresh_console**(*image*, *console*)

tcod.**image_rotate90**(*image*, *num=1*)

tcod.**image_save**(*image*, *filename*)

tcod.**image_scale**(*image*, *neww*, *newh*)

tcod.**image_set_key_color**(*image*, *col*)

tcod.**image_vflip**(*image*)

line

tcod.**line_init**(*xo*, *yo*, *xd*, *yd*)

Initilize a line whose points will be returned by *line_step*.

This function does not return anything on its own.

Does not include the origin point.

Parameters

- **xo** (*int*) – X starting point.
- **yo** (*int*) – Y starting point.
- **xd** (*int*) – X destination point.
- **yd** (*int*) – Y destination point.

Deprecated since version 2.0: Use *line_iter* instead.

tcod.**line_step**()

After calling line_init returns (x, y) points of the line.

Once all points are exhausted this function will return (None, None)

Returns The next (x, y) point of the line setup by line_init, or (None, None) if there are no more points.

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

Deprecated since version 2.0: Use *line_iter* instead.

tcod.**line**(*xo*, *yo*, *xd*, *yd*, *py_callback*)

Iterate over a line using a callback function.

Your callback function will take x and y parameters and return True to continue iteration or False to stop iteration and return.

This function includes both the start and end points.

Parameters

- **xo** (*int*) – X starting point.
- **yo** (*int*) – Y starting point.
- **xd** (*int*) – X destination point.
- **yd** (*int*) – Y destination point.
- **py_callback** (*Callable[[int, int], bool]*) – A callback which takes x and y parameters and returns bool.

Returns

False if the callback cancels the line interation by returning False or None, otherwise True.

Return type bool

Deprecated since version 2.0: Use *line_iter* instead.

tcod.**line_iter**(*xo*, *yo*, *xd*, *yd*)

returns an iterator

This iterator does not include the origin point.

Parameters

- **xo** (*int*) – X starting point.
- **yo** (*int*) – Y starting point.
- **xd** (*int*) – X destination point.
- **yd** (*int*) – Y destination point.

Returns An iterator of (x,y) points.

Return type Iterator[Tuple[int,int]]

map

tcod.**map_clear**(*m*, *walkable=False*, *transparent=False*)

tcod.**map_compute_fov**(*m*, *x*, *y*, *radius=0*, *light_walls=True*, *algo=12*)

tcod.**map_copy**(*source*, *dest*)

tcod.**map_delete**(*m*)

tcod.**map_get_height**(*map*)

tcod.**map_get_width**(*map*)

tcod.**map_is_in_fov**(*m*, *x*, *y*)

tcod.**map_is_transparent**(*m*, *x*, *y*)

tcod.**map_is_walkable**(*m*, *x*, *y*)

tcod.**map_new**(*w*, *h*)

tcod.**map_set_properties**(*m*, *x*, *y*, *isTrans*, *isWalk*)

mouse

tcod.**mouse_get_status**()

tcod.**mouse_is_cursor_visible**()

tcod.**mouse_move**(*x*, *y*)

tcod.**mouse_show_cursor**(*visible*)

namegen

tcod.**namegen_destroy**()

tcod.**namegen_generate**(*name*)

tcod.**namegen_generate_custom**(*name*, *rule*)

tcod.**namegen_get_sets**()

tcod.**namegen_parse**(*filename*, *random=None*)

noise

tcod.**noise_delete**(*n*)
Does nothing.

tcod.**noise_get**(*n*, *f*, *typ=0*)
Return the noise value sampled from the `f` coordinate.

`f` should be a tuple or list with a length matching `Noise.dimensions`. If `f` is shoerter than `Noise.dimensions` the missing coordinates will be filled with zeros.

Parameters

- **n** (`Noise`) – A Noise instance.
- **f** (*`Sequence[float]`*) – The point to sample the noise from.
- **typ** (*`int`*) – The noise algorithm to use.

Returns The sampled noise value.

Return type float

tcod.**noise_get_fbm**(*n*, *f*, *oc*, *typ=0*)
Return the fractal Brownian motion sampled from the `f` coordinate.

Parameters

- **n** (Noise) – A Noise instance.
- **f** (*Sequence[float]*) – The point to sample the noise from.
- **typ** (*int*) – The noise algorithm to use.
- **octaves** (*float*) – The level of level. Should be more than 1.

Returns The sampled noise value.

Return type float

tcod.**noise_get_turbulence**(*n*, *f*, *oc*, *typ=0*)

Return the turbulence noise sampled from the `f` coordinate.

Parameters

- **n** (Noise) – A Noise instance.
- **f** (*Sequence[float]*) – The point to sample the noise from.
- **typ** (*int*) – The noise algorithm to use.
- **octaves** (*float*) – The level of level. Should be more than 1.

Returns The sampled noise value.

Return type float

tcod.**noise_new**(*dim*, *h=0.5*, *l=2.0*, *random=None*)

Return a new Noise instance.

Parameters

- **dim** (*int*) – Number of dimensions. From 1 to 4.
- **h** (*float*) – The hurst exponent. Should be in the 0.0-1.0 range.
- **l** (*float*) – The noise lacunarity.
- **random** (*Optional[Random]*) – A Random instance, or None.

Returns The new Noise instance.

Return type *Noise*

tcod.**noise_set_type**(*n*, *typ*)

Set a Noise objects default noise algorithm.

Parameters **typ** (*int*) – Any NOISE_* constant.

parser

tcod.**parser_delete**(*parser*)

tcod.**parser_get_bool_property**(*parser*, *name*)

tcod.**parser_get_char_property**(*parser*, *name*)

tcod.**parser_get_color_property**(*parser*, *name*)

tcod.**parser_get_dice_property**(*parser*, *name*)

tcod.**parser_get_float_property**(*parser*, *name*)

tcod.**parser_get_int_property**(*parser*, *name*)

tcod.**parser_get_list_property**(*parser*, *name*, *type*)

tcod.**parser_get_string_property**(*parser*, *name*)

tcod.**parser_new**()

tcod.**parser_new_struct**(*parser*, *name*)

tcod.**parser_run**(*parser*, *filename*, *listener=None*)

random

tcod.**random_delete**(*rnd*)
Does nothing.

tcod.**random_get_double**(*rnd*, *mi*, *ma*)
Return a random float in the range: mi <= n <= ma.

Deprecated since version 2.0: Use *random_get_float* instead. Both funtions return a double precision float.

tcod.**random_get_double_mean**(*rnd*, *mi*, *ma*, *mean*)
Return a random weighted float in the range: mi <= n <= ma.

Deprecated since version 2.0: Use *random_get_float_mean* instead. Both funtions return a double precision float.

tcod.**random_get_float**(*rnd*, *mi*, *ma*)
Return a random float in the range: mi <= n <= ma.

The result is affacted by calls to *random_set_distribution*.

Parameters

- **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.
- **low** (*float*) – The lower bound of the random range, inclusive.
- **high** (*float*) – The upper bound of the random range, inclusive.

Returns

A random double precision float in the range mi <= n <= ma.

Return type float

tcod.**random_get_float_mean**(*rnd*, *mi*, *ma*, *mean*)
Return a random weighted float in the range: mi <= n <= ma.

The result is affacted by calls to *random_set_distribution*.

Parameters

- **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.
- **low** (*float*) – The lower bound of the random range, inclusive.
- **high** (*float*) – The upper bound of the random range, inclusive.
- **mean** (*float*) – The mean return value.

Returns

A random weighted double precision float in the range mi <= n <= ma.

Return type float

tcod.**random_get_instance**()
Return the default Random instance.

Returns A Random instance using the default random number generator.

Return type *Random*

tcod.**random_get_int**(*rnd*, *mi*, *ma*)
Return a random integer in the range: `mi` <= n <= `ma`.

The result is affacted by calls to *random_set_distribution*.

Parameters

- **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.
- **low** (*int*) – The lower bound of the random range, inclusive.
- **high** (*int*) – The upper bound of the random range, inclusive.

Returns A random integer in the range `mi` <= n <= `ma`.

Return type int

tcod.**random_get_int_mean**(*rnd*, *mi*, *ma*, *mean*)
Return a random weighted integer in the range: `mi` <= n <= `ma`.

The result is affacted by calls to *random_set_distribution*.

Parameters

- **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.
- **low** (*int*) – The lower bound of the random range, inclusive.
- **high** (*int*) – The upper bound of the random range, inclusive.
- **mean** (*int*) – The mean return value.

Returns A random weighted integer in the range `mi` <= n <= `ma`.

Return type int

tcod.**random_new**(*algo=1*)
Return a new Random instance. Using `algo`.

Parameters **algo** (*int*) – The random number algorithm to use.

Returns A new Random instance using the given algorithm.

Return type *Random*

tcod.**random_new_from_seed**(*seed*, *algo=1*)
Return a new Random instance. Using the given `seed` and `algo`.

Parameters

- **seed** (*Hashable*) – The RNG seed. Should be a 32-bit integer, but any hashable object is accepted.
- **algo** (*int*) – The random number algorithm to use.

Returns A new Random instance using the given algorithm.

Return type *Random*

tcod.**random_restore**(*rnd*, *backup*)
Restore a random number generator from a backed up copy.

Parameters

- **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.
- **backup** (Random) – The Random instance which was used as a backup.

tcod.**random_save**(*rnd*)
Return a copy of a random number generator.

Parameters **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.

Returns A Random instance with a copy of the random generator.

Return type *Random*

tcod.**random_set_distribution**(*rnd*, *dist*)
Change the distribution mode of a random number generator.

Parameters

- **rnd** (*Optional[Random]*) – A Random instance, or None to use the default.
- **dist** (*int*) – The distribution mode to use. Should be DISTRIBUTION_*.

struct

tcod.**struct_add_flag**(*struct*, *name*)

tcod.**struct_add_list_property**(*struct*, *name*, *typ*, *mandatory*)

tcod.**struct_add_property**(*struct*, *name*, *typ*, *mandatory*)

tcod.**struct_add_structure**(*struct*, *sub_struct*)

tcod.**struct_add_value_list**(*struct*, *name*, *value_list*, *mandatory*)

tcod.**struct_get_name**(*struct*)

tcod.**struct_get_type**(*struct*, *name*)

tcod.**struct_is_mandatory**(*struct*, *name*)

other

class tcod.**ConsoleBuffer**(*width*, *height*, *back_r=0*, *back_g=0*, *back_b=0*, *fore_r=0*, *fore_g=0*, *fore_b=0*, *char=' '*)
Simple console that allows direct (fast) access to cells. simplifies use of the "fill" functions.

Parameters

- **width** (*int*) – Width of the new ConsoleBuffer.
- **height** (*int*) – Height of the new ConsoleBuffer.
- **back_r** (*int*) – Red background color, from 0 to 255.
- **back_g** (*int*) – Green background color, from 0 to 255.
- **back_b** (*int*) – Blue background color, from 0 to 255.

- **fore_r** (*int*) – Red foreground color, from 0 to 255.
- **fore_g** (*int*) – Green foreground color, from 0 to 255.
- **fore_b** (*int*) – Blue foreground color, from 0 to 255.
- **char** (*AnyStr*) – A single character str or bytes object.

blit(*dest*, *fill_fore=True*, *fill_back=True*)
Use libtcod's "fill" functions to write the buffer to a console.

Parameters

- **dest** (*Console*) – Console object to modify.
- **fill_fore** (*bool*) – If True, fill the foreground color and characters.
- **fill_back** (*bool*) – If True, fill the background color.

clear(*back_r=0*, *back_g=0*, *back_b=0*, *fore_r=0*, *fore_g=0*, *fore_b=0*, *char=' '*)
Clears the console. Values to fill it with are optional, defaults to black with no characters.

Parameters

- **back_r** (*int*) – Red background color, from 0 to 255.
- **back_g** (*int*) – Green background color, from 0 to 255.
- **back_b** (*int*) – Blue background color, from 0 to 255.
- **fore_r** (*int*) – Red foreground color, from 0 to 255.
- **fore_g** (*int*) – Green foreground color, from 0 to 255.
- **fore_b** (*int*) – Blue foreground color, from 0 to 255.
- **char** (*AnyStr*) – A single character str or bytes object.

copy()
Returns a copy of this ConsoleBuffer.

Returns A new ConsoleBuffer copy.

Return type *ConsoleBuffer*

set(*x*, *y*, *back_r*, *back_g*, *back_b*, *fore_r*, *fore_g*, *fore_b*, *char*)
Set the background color, foreground color and character of one cell.

Parameters

- **x** (*int*) – X position to change.
- **y** (*int*) – Y position to change.
- **back_r** (*int*) – Red background color, from 0 to 255.
- **back_g** (*int*) – Green background color, from 0 to 255.
- **back_b** (*int*) – Blue background color, from 0 to 255.
- **fore_r** (*int*) – Red foreground color, from 0 to 255.
- **fore_g** (*int*) – Green foreground color, from 0 to 255.
- **fore_b** (*int*) – Blue foreground color, from 0 to 255.
- **char** (*AnyStr*) – A single character str or bytes object.

set_back(*x*, *y*, *r*, *g*, *b*)
Set the background color of one cell.

Parameters

- **x** (*int*) – X position to change.
- **y** (*int*) – Y position to change.
- **r** (*int*) – Red background color, from 0 to 255.
- **g** (*int*) – Green background color, from 0 to 255.
- **b** (*int*) – Blue background color, from 0 to 255.
- **char** (*AnyStr*) – A single character str or bytes object.

set_fore(*x*, *y*, *r*, *g*, *b*, *char*)

Set the character and foreground color of one cell.

Parameters

- **x** (*int*) – X position to change.
- **y** (*int*) – Y position to change.
- **r** (*int*) – Red foreground color, from 0 to 255.
- **g** (*int*) – Green foreground color, from 0 to 255.
- **b** (*int*) – Blue foreground color, from 0 to 255.
- **char** (*AnyStr*) – A single character str or bytes object.

class tcod.**Dice**(*nb_dices=0*, *nb_faces=0*, *multiplier=0*, *addsub=0*)

Parameters

- **nb_dices** (*int*) – Number of dice.
- **nb_faces** (*int*) – Number of sides on a die.
- **multiplier** (*float*) – Multiplier.
- **addsub** (*float*) – Addition.

Deprecated since version 2.0: You should make your own dice functions instead of using this class which is tied to a CData object.

CHAPTER 17

tdl

Getting Started

Once the library is imported you can load the font you want to use with *tdl.set_font*. This is optional and when skipped will use a decent default font.

After that you call *tdl.init* to set the size of the window and get the root console in return. This console is the canvas to what will appear on the screen.

Indexing Consoles

For most methods taking a position you can use Python-style negative indexes to refer to the opposite side of a console with (-1, -1) starting at the bottom right. You can also check if a point is part of a console using containment logic i.e. ((x, y) in console).

You may also iterate over a console using a for statement. This returns every x,y coordinate available to draw on but it will be extremely slow to actually operate on every coordinate individualy. Try to minimize draws by using an offscreen *Console*, only drawing what needs to be updated, and using *Console.blit*.

Drawing and Colors

Once you have the root console from *tdl.init* you can start drawing on it using a method such as *Console.draw_char*. When using this method you can have the char parameter be an integer or a single character string.

The fg and bg parameters expect a variety of types. The parameters default to Ellipsis which will tell the function to use the colors previously set by the *Console.set_colors* method. The colors set by :any‘Console.set_colors‘ are per each *Console*/*Window* and default to white on black. You can use a

3-item list/tuple of [red, green, blue] with integers in the 0-255 range with [0, 0, 0] being black and [255, 255, 255] being white. You can even use a single integer of 0xRRGGBB if you like.

Using None in the place of any of the three parameters (char, fg, bg) will tell the function to not overwrite that color or character.

After the drawing functions are called a call to `tdl.flush` will update the screen.

tdl API

tdl.**set_font**(*path*, *columns=None*, *rows=None*, *columnFirst=False*, *greyscale=False*, *altLayout=False*)
Changes the font to be used for this session. This should be called before `tdl.init`

If the font specifies its size in its filename (i.e. font_NxN.png) then this function can auto-detect the tileset formatting and the parameters columns and rows can be left None.

While it's possible you can change the font mid program it can sometimes break in rare circumstances. So use caution when doing this.

Parameters

- **path** (*Text*) – A file path to a *.bmp* or *.png* file.
- **columns** (*int*) – Number of columns in the tileset.

  Can be left None for auto-detection.
- **rows** (*int*) – Number of rows in the tileset.

  Can be left None for auto-detection.
- **columnFirst** (*bool*) – Defines if the characer order goes along the rows or colomns.

  It should be True if the charater codes 0-15 are in the first column, and should be False if the characters 0-15 are in the first row.
- **greyscale** (*bool*) – Creates an anti-aliased font from a greyscale bitmap. Otherwise it uses the alpha channel for anti-aliasing.

  Unless you actually need anti-aliasing from a font you know uses a smooth greyscale channel you should leave this on False.
- **altLayout** (*bool*) – An alternative layout with space in the upper left corner. The colomn parameter is ignored if this is True, find examples of this layout in the *font/libtcod/* directory included with the python-tdl source.

Raises *TDLError* – Will be raised if no file is found at path or if auto- detection fails.

tdl.**init**(*width*, *height*, *title=None*, *fullscreen=False*, *renderer=u'SDL'*)
Start the main console with the given width and height and return the root console.

Call the consoles drawing functions. Then remember to use L{tdl.flush} to make what's drawn visible on the console.

Parameters

- **width** (*int*) – width of the root console (in tiles)
- **height** (*int*) – height of the root console (in tiles)
- **title** (*Optiona[Text]*) – Text to display as the window title.

  If left None it defaults to the running scripts filename.

- **fullscreen** (*bool*) – Can be set to True to start in fullscreen mode.
- **renderer** (*Text*) – Can be one of 'GLSL', 'OPENGL', or 'SDL'.
- **to way Python works you're unlikely to see much of an** (*Due*) –
- **by using 'GLSL' over 'OPENGL' as most of the** (*improvement*) –
- **Python is slow interacting with the console and the** (*time*) –
- **itself is pretty fast even on 'SDL'.** (*rendering*) –

Returns

The root console.

Only what is drawn on the root console is what's visible after a call to *tdl.flush*. After the root console is garbage collected, the window made by this function will close.

Return type *tdl.Console*

See also:

Console set_font

tdl.**flush**()
Make all changes visible and update the screen.

Remember to call this function after drawing operations. Calls to flush will enfore the frame rate limit set by *tdl.set_fps*.

This function can only be called after *tdl.init*

tdl.**screenshot**(*path=None*)
Capture the screen and save it as a png file.

If path is None then the image will be placed in the current folder with the names: `screenshot001.png, screenshot002.png, ...`

Parameters **path** (*Optional[Text]*) – The file path to save the screenshot.

tdl.**get_fullscreen**()
Returns True if program is fullscreen.

Returns

Returns True if the application is in full-screen mode. Otherwise returns False.

Return type bool

tdl.**set_fullscreen**(*fullscreen*)
Changes the fullscreen state.

Parameters **fullscreen** (*bool*) – True for full-screen, False for windowed mode.

tdl.**set_title**(*title*)
Change the window title.

Parameters **title** (*Text*) – The new title text.

tdl.**get_fps**()
Return the current frames per second of the running program set by *set_fps*

Returns

The frame rate set by *set_fps*. If there is no current limit, this will return 0.

Return type int

tdl.**set_fps**(*fps*)
Set the maximum frame rate.

Further calls to *tdl.flush* will limit the speed of the program to run at *fps* frames per second. This can also be set to None to remove the frame rate limit.

Parameters **fps** (*optional[int]*) – The frames per second limit, or None.

tdl.**force_resolution**(*width*, *height*)
Change the fullscreen resoulution.

Parameters

- **width** (*int*) – Width in pixels.
- **height** (*int*) – Height in pixels.

exception tdl.**TDLError**
The catch all for most TDL specific errors.

tdl.Console

class tdl.**Console**(*width*, *height*)
Contains character and color data and can be drawn to.

The console created by the *tdl.init* function is the root console and is the console that is rendered to the screen with *flush*.

Any console created from the Console class is an off-screen console that can be drawn on before being *blit* to the root console.

Parameters

- **width** (*int*) – Width of the new console in tiles
- **height** (*int*) – Height of the new console in tiles

__del__()
If the main console is garbage collected then the window will be closed as well

__format__()
default object formatter

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__sizeof__() → int
size of object in memory, in bytes

blit(*source*, *x=0*, *y=0*, *width=None*, *height=None*, *srcX=0*, *srcY=0*, *fg_alpha=1.0*, *bg_alpha=1.0*)
Blit another console or Window onto the current console.

By default it blits the entire source to the topleft corner.

Parameters

- **source** (*Union[tdl.Console, tdl.Window]*) – The blitting source. A console can blit to itself without any problems.
- **x** (*int*) – x-coordinate of this console to blit on.

- **y** (*int*) – y-coordinate of this console to blit on.
- **width** (*Optional[int]*) – Width of the rectangle.

  Can be None to extend as far as possible to the bottom right corner of the blit area or can be a negative number to be sized reltive to the total size of the B{destination} console.
- **height** (*Optional[int]*) – Height of the rectangle.
- **srcX** (*int*) – x-coordinate of the source region to blit.
- **srcY** (*int*) – y-coordinate of the source region to blit.
- **fg_alpha** (*float*) – The foreground alpha.

clear (*fg=Ellipsis*, *bg=Ellipsis*)
Clears the entire L{Console}/L{Window}.

Unlike other drawing functions, fg and bg can not be None.

Parameters

- **fg** (*Union[Tuple[int, int, int], int, Ellipsis]*) –
- **bg** (*Union[Tuple[int, int, int], int, Ellipsis]*) –

See also:

draw_rect

draw_char (*x*, *y*, *char*, *fg=Ellipsis*, *bg=Ellipsis*)
Draws a single character.

Parameters

- **x** (*int*) – x-coordinate to draw on.
- **y** (*int*) – y-coordinate to draw on.
- **char** (*Optional[Union[int, Text]]*) – An integer, single character string, or None.

  You can set the char parameter as None if you only want to change the colors of the tile.
- **fg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –
- **bg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –

Raises: AssertionError: Having x or y values that can't be placed inside of the console will raise an AssertionError. You can use always use ((x, y) in console) to check if a tile is drawable.

See also:

get_char

draw_frame (*x*, *y*, *width*, *height*, *string*, *fg=Ellipsis*, *bg=Ellipsis*)
Similar to L{draw_rect} but only draws the outline of the rectangle.

width or 'height can be None to extend to the bottom right of the console or can be a negative number to be sized reltive to the total size of the console.

Parameters

- **x** (*int*) – The x-coordinate to start on.
- **y** (*int*) – The y-coordinate to start on.
- **width** (*Optional[int]*) – Width of the rectangle.

- **height** (*Optional[int]*) – Height of the rectangle.
- **string** (*Optional[Union[Text, int]]*) – An integer, single character string, or None.

  You can set this parameter as None if you only want to change the colors of an area.
- **fg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –
- **bg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –

Raises

AssertionError – Having x or y values that can't be placed inside of the console will raise an AssertionError.

You can use always use `((x, y) in console)` to check if a tile is drawable.

See also:

draw_rect, *Window*

draw_rect(*x*, *y*, *width*, *height*, *string*, *fg=Ellipsis*, *bg=Ellipsis*)

Draws a rectangle starting from x and y and extending to width and height.

If width or height are None then it will extend to the edge of the console.

Parameters

- **x** (*int*) – x-coordinate for the top side of the rect.
- **y** (*int*) – y-coordinate for the left side of the rect.
- **width** (*Optional[int]*) – The width of the rectangle.

  Can be None to extend to the bottom right of the console or can be a negative number to be sized reltive to the total size of the console.
- **height** (*Optional[int]*) – The height of the rectangle.
- **string** (*Optional[Union[Text, int]]*) – An integer, single character string, or None.

  You can set the string parameter as None if you only want to change the colors of an area.
- **fg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –
- **bg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –

Raises

- `AssertionError` – Having x or y values that can't be placed inside of the console will raise an AssertionError.
- You can use always use `((x, y) in console)` to check if a tile
- is drawable.

See also:

clear, *draw_frame*

draw_str(*x*, *y*, *string*, *fg=Ellipsis*, *bg=Ellipsis*)

Draws a string starting at x and y.

A string that goes past the right side will wrap around. A string wrapping to below the console will raise *tdl.TDLError* but will still be written out. This means you can safely ignore the errors with a try..except block if you're fine with partially written strings.

r and n are drawn on the console as normal character tiles. No special encoding is done and any string will translate to the character table as is.

For a string drawing operation that respects special characters see *print_str*.

Parameters

- **x** (*int*) – x-coordinate to start at.
- **y** (*int*) – y-coordinate to start at.
- **string** (*Union[Text, Iterable[int]]*) – A string or an iterable of numbers.

  Special characters are ignored and rendered as any other character.
- **fg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –
- **bg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –

Raises

AssertionError – Having x or y values that can't be placed inside of the console will raise an AssertionError.

You can use always use `((x, y) in console)` to check if a tile is drawable.

See also:

print_str

get_char(*x*, *y*)

Return the character and colors of a tile as (ch, fg, bg)

This method runs very slowly as is not recommended to be called frequently.

Parameters

- **x** (*int*) – The x-coordinate to pick.
- **y** (*int*) – The y-coordinate to pick.

Returns

A 3-item tuple: *(int, fg, bg)*

The first item is an integer of the character at the position (x, y) the second and third are the foreground and background colors respectfully.

Return type Tuple[int, Tuple[int, int, int], Tuple[int, int, int]]

See also:

draw_char

get_cursor()

Return the virtual cursor position.

The cursor can be moved with the *move* method.

Returns

The (x, y) coordinate of where *print_str* will continue from.

Return type Tuple[int, int]

See also:

:any:move‘

get_size()
Return the size of the console as (width, height)

Returns A (width, height) tuple.

Return type Tuple[int, int]

move(*x*, *y*)
Move the virtual cursor.

Parameters

- **x** (*int*) – x-coordinate to place the cursor.
- **y** (*int*) – y-coordinate to place the cursor.

See also:

get_cursor, *print_str*, *write*

print_str(*string*)
Print a string at the virtual cursor.

Handles special characters such as 'n' and 'r'. Printing past the bottom of the console will scroll everything upwards if *set_mode* is set to 'scroll'.

Colors can be set with *set_colors* and the virtual cursor can be moved with *move*.

Parameters **string** (*Text*) – The text to print.

See also:

draw_str, *move*, *set_colors*, *set_mode*, *write*, *Window*

scroll(*x*, *y*)
Scroll the contents of the console in the direction of x,y.

Uncovered areas will be cleared to the default background color. Does not move the virutal cursor.

Parameters

- **x** (*int*) – Distance to scroll along the x-axis.
- **y** (*int*) – Distance to scroll along the y-axis.

Returns An iterator over the (x, y) coordinates of any tile uncovered after scrolling.

Return type Iterator[Tuple[int, int]]

See also:

set_colors

set_colors(*fg=None*, *bg=None*)
Sets the colors to be used with the L{print_str} and draw_* methods.

Values of None will only leave the current values unchanged.

Parameters

- **fg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –
- **bg** (*Optional[Union[Tuple[int, int, int], int, Ellipsis]]*) –

See also:

move, *print_str*

set_mode(*mode*)
Configure how this console will react to the cursor writing past the end if the console.

This is for methods that use the virtual cursor, such as *print_str*.

Parameters

- **mode** (*Text*) – The mode to set.
- **settings are** (*Possible*) –
  - 'error' - A TDLError will be raised once the cursor reaches the end of the console. Everything up until the error will still be drawn. This is the default setting.
  - 'scroll' - The console will scroll up as stuff is written to the end. You can restrict the region with *tdl.Window* when doing this.

..seealso:: *write*, *print_str*

write(*string*)
This method mimics basic file-like behaviour.

Because of this method you can replace sys.stdout or sys.stderr with a *Console* or *Window* instance.

This is a convoluted process and behaviour seen now can be excepted to change on later versions.

Parameters **string** (*Text*) – The text to write out.

See also:

set_colors, *set_mode*, *Window*

__delattr__
x.__delattr__('name') <==> del x.name

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__str__

tdl.Window

class tdl.**Window**(*console*, *x*, *y*, *width*, *height*)
Isolate part of a *Console* or *Window* instance.

This classes methods are the same as *tdl.Console*

Making a Window and setting its width or height to None will extend it to the edge of the console.

This follows the normal rules for indexing so you can use a negative integer to place the Window relative to the bottom right of the parent Console instance.

width or *height* can be set to None to extend as far as possible to the bottom right corner of the parent Console or can be a negative number to be sized reltive to the Consoles total size.

Parameters

- **console** (*Union(tdl.Console, tdl.Window)*) – The parent object.
- **x** (*int*) – x-coordinate to place the Window.

- **y** (*int*) – y-coordinate to place the Window.
- **width** (*Optional[int]*) – Width of the Window.
- **height** (*Optional[int]*) – Height of the Window.

CHAPTER 18

tdl.event

This module handles user input.

To handle user input you will likely want to use the *event.get* function or create a subclass of *event.App*.

- *tdl.event.get* iterates over recent events.
- *tdl.event.App* passes events to the overridable methods: `ev_*` and `key_*`.

But there are other options such as *event.key_wait* and *event.is_window_closed*.

A few event attributes are actually string constants. Here's a reference for those:

- *Event.type*: 'QUIT', 'KEYDOWN', 'KEYUP', 'MOUSEDOWN', 'MOUSEUP', or 'MOUSEMOTION.'
- *MouseButtonEvent.button* (found in *MouseDown* and *MouseUp* events): 'LEFT', 'MIDDLE', 'RIGHT', 'SCROLLUP', 'SCROLLDOWN'
- *KeyEvent.key* (found in *KeyDown* and *KeyUp* events): 'NONE', 'ESCAPE', 'BACKSPACE', 'TAB', 'ENTER', 'SHIFT', 'CONTROL', 'ALT', 'PAUSE', 'CAPSLOCK', 'PAGEUP', 'PAGEDOWN', 'END', 'HOME', 'UP', 'LEFT', 'RIGHT', 'DOWN', 'PRINTSCREEN', 'INSERT', 'DELETE', 'LWIN', 'RWIN', 'APPS', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'KP0', 'KP1', 'KP2', 'KP3', 'KP4', 'KP5', 'KP6', 'KP7', 'KP8', 'KP9', 'KPADD', 'KPSUB', 'KPDIV', 'KPMUL', 'KPDEC', 'KPENTER', 'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9', 'F10', 'F11', 'F12', 'NUMLOCK', 'SCROLLLOCK', 'SPACE', 'CHAR'

class tdl.event.**Quit**
Fired when the window is closed by the user.

class tdl.event.**MouseUp**(*button*, *pos*, *cell*)
Fired when a mouse button is released.

class tdl.event.**KeyDown**(*key=''*, *char=''*, *text=''*, *shift=False*, *left_alt=False*, *right_alt=False*, *left_control=False*, *right_control=False*, *left_meta=False*, *right_meta=False*)
Fired when the user presses a key on the keyboard or a key repeats.

class tdl.event.**MouseMotion**(*pos*, *cell*, *motion*, *cellmotion*)
Fired when the mouse is moved.

cell = None
(x, y) position of the mouse snapped to a cell on the root console. type: (int, int)

cellmotion = None
(x, y) mostion of the mouse moving over cells on the root console. type: (int, int)

motion = None
(x, y) motion of the mouse on the screen. type: (int, int)

pos = None
(x, y) position of the mouse on the screen. type: (int, int)

class tdl.event.**MouseButtonEvent**(*button*, *pos*, *cell*)
Base class for mouse button events.

button = None
Text – Can be one of 'LEFT', 'MIDDLE', 'RIGHT', 'SCROLLUP', 'SCROLLDOWN'

cell = None
Tuple[int, int] – (x, y) position of the mouse snapped to a cell on the root console

pos = None
Tuple[int, int] – (x, y) position of the mouse on the screen.

class tdl.event.**MouseDown**(*button*, *pos*, *cell*)
Fired when a mouse button is pressed.

class tdl.event.**KeyEvent**(*key=''*, *char=''*, *text=''*, *shift=False*, *left_alt=False*, *right_alt=False*, *left_control=False*, *right_control=False*, *left_meta=False*, *right_meta=False*)
Base class for key events.

alt = None
bool – True if alt was held down during this event.

char = None
Text – A single character string of the letter or symbol pressed.

Special characters like delete and return are not cross-platform. L{key} or L{keychar} should be used instead for special keys. Characters are also case sensitive.

control = None
bool – True if control was held down during this event.

key = None
Text – Human readable names of the key pressed. Non special characters will show up as 'CHAR'.

Can be one of 'NONE', 'ESCAPE', 'BACKSPACE', 'TAB', 'ENTER', 'SHIFT', 'CONTROL', 'ALT', 'PAUSE', 'CAPSLOCK', 'PAGEUP', 'PAGEDOWN', 'END', 'HOME', 'UP', 'LEFT', 'RIGHT', 'DOWN', 'PRINTSCREEN', 'INSERT', 'DELETE', 'LWIN', 'RWIN', 'APPS', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'KP0', 'KP1', 'KP2', 'KP3', 'KP4', 'KP5', 'KP6', 'KP7', 'KP8', 'KP9', 'KPADD', 'KPSUB', 'KPDIV', 'KPMUL', 'KPDEC', 'KPENTER', 'F1', 'F2', 'F3', 'F4', 'F5', 'F6', 'F7', 'F8', 'F9', 'F10', 'F11', 'F12', 'NUMLOCK', 'SCROLLLOCK', 'SPACE', 'CHAR'

For the actual character instead of 'CHAR' use *keychar*.

keychar = None
Similar to L{key} but returns a case sensitive letter or symbol instead of 'CHAR'.

This variable makes available the widest variety of symbols and should be used for key-mappings or anywhere where a narrower sample of keys isn't needed.

leftAlt = None
bool

leftCtrl = None
bool

left_alt = None
bool

left_control = None
bool

rightAlt = None
bool

rightCtrl = None
bool

right_alt = None
bool

right_control = None
bool

shift = None
bool – True if shift was held down during this event.

class tdl.event.**App**
Application framework.

- ev_*: Events are passed to methods based on their *Event.type* attribute. If an event type is 'KEYDOWN' the ev_KEYDOWN method will be called with the event instance as a parameter.
- key_*: When a key is pressed another method will be called based on the *KeyEvent.key* attribute. For example the 'ENTER' key will call key_ENTER with the associated *KeyDown* event as its parameter.
- *update*: This method is called every loop. It is passed a single parameter detailing the time in seconds since the last update (often known as deltaTime.)

  You may want to call drawing routines in this method followed by *tdl.flush*.

ev_KEYDOWN(*event*)
Override this method to handle a *KeyDown* event.

ev_KEYUP(*event*)
Override this method to handle a *KeyUp* event.

ev_MOUSEDOWN(*event*)
Override this method to handle a *MouseDown* event.

ev_MOUSEMOTION(*event*)
Override this method to handle a *MouseMotion* event.

ev_MOUSEUP(*event*)
Override this method to handle a *MouseUp* event.

ev_QUIT(*event*)
Unless overridden this method raises a SystemExit exception closing the program.

run()
Delegate control over to this App instance. This function will process all events and send them to the special methods ev_* and key_*.

A call to *App.suspend* will return the control flow back to where this function is called. And then the App can be run again. But a single App instance can not be run multiple times simultaneously.

run_once()
Pump events to this App instance and then return.

This works in the way described in *App.run* except it immediately returns after the first *update* call.

Having multiple *App* instances and selectively calling runOnce on them is a decent way to create a state machine.

suspend()
When called the App will begin to return control to where *App.run* was called.

Some further events are processed and the *App.update* method will be called one last time before exiting (unless suspended during a call to *App.update*.)

update(*deltaTime*)
Override this method to handle per frame logic and drawing.

Parameters **deltaTime** (*float*) – This parameter tells the amount of time passed since the last call measured in seconds as a floating point number.

You can use this variable to make your program frame rate independent. Use this parameter to adjust the speed of motion, timers, and other game logic.

class tdl.event.**KeyUp**(*key=''*, *char=''*, *text=''*, *shift=False*, *left_alt=False*, *right_alt=False*, *left_control=False*, *right_control=False*, *left_meta=False*, *right_meta=False*)
Fired when the user releases a key on the keyboard.

class tdl.event.**Event**
Base Event class.

You can easily subclass this to make your own events. Be sure to set the class attribute L{Event.type} for it to be passed to a custom *App* ev_* method.

__repr__()
List an events public attributes when printed.

type = None
String constant representing the type of event.

The *App* ev_* methods depend on this attribute.

Can be: 'QUIT', 'KEYDOWN', 'KEYUP', 'MOUSEDOWN', 'MOUSEUP', or 'MOUSEMOTION.'

tdl.event.**key_wait**()
Waits until the user presses a key. Then returns a *KeyDown* event.

Key events will repeat if held down.

A click to close the window will be converted into an Alt+F4 KeyDown event.

Returns The pressed key.

Return type *tdl.event.KeyDown*

tdl.event.**set_key_repeat**(*delay=500*, *interval=0*)
Does nothing.

tdl.event.**get**()
Flushes the event queue and returns the list of events.

This function returns *Event* objects that can be identified by their type attribute or their class.

Returns: Iterator[Type[Event]]: An iterable of Events or anything put in a *push* call.

If the iterator is deleted or otherwise interrupted before finishing the excess items are preserved for the next call.

tdl.event.**wait**(*timeout=None*, *flush=True*)
Wait for an event.

Parameters

- **timeout** (*Optional[int]*) – The time in seconds that this function will wait before giving up and returning None.

  With the default value of None, this will block forever.

- **flush** (*bool*) – If True a call to *tdl.flush* will be made before listening for events.

Returns: Type[Event]: An event, or None if the function has timed out. Anything added via *push* will also be returned.

tdl.event.**is_window_closed**()

Returns True if the exit button on the window has been clicked and stays True afterwards.

Returns: bool:

tdl.event.**push**(*event*)

Push an event into the event buffer.

Parameters **event** (*Any*) – This event will be available on the next call to *event.get*.

An event pushed in the middle of a *get* will not show until the next time *get* called preventing push related infinite loops.

This object should at least have a 'type' attribute.

CHAPTER 19

tdl.map

Rogue-like map utilitys such as line-of-sight, field-of-view, and path-finding.

Deprecated since version 3.2: The features provided here are better realized in the *tcod.map* and *tcod.path* modules.

class tdl.map.**AStar**(*width*, *height*, *callback*, *diagnalCost=1.4142135623730951*, *advanced=False*)

An A* pathfinder using a callback.

Deprecated since version 3.2: See *tcod.path*.

Before crating this instance you should make one of two types of callbacks:

•A function that returns the cost to move to (x, y)

•A function that returns the cost to move between (destX, destY, sourceX, sourceY)

If path is blocked the function should return zero or None. When using the second type of callback be sure to set advanced=True

Parameters

- **width** (*int*) – Width of the pathfinding area (in tiles.)
- **height** (*int*) – Height of the pathfinding area (in tiles.)
- **(Union[Callable[[int, int],** **float],** (*callback*) – Callable[[int, int, int, int], float]]): A callback returning the cost of a tile or edge.

  A callback taking parameters depending on the setting of 'advanced' and returning the cost of movement for an open tile or zero for a blocked tile.
- **diagnalCost** (*float*) – Multiplier for diagonal movement.

  Can be set to zero to disable diagonal movement entirely.
- **advanced** (*bool*) – Give 2 additional parameters to the callback.

  A simple callback with 2 positional parameters may not provide enough information. Setting this to True will call the callback with 2 additional parameters giving you both the destination and the source of movement.

When True the callback will need to accept (destX, destY, sourceX, sourceY) as parameters. Instead of just (destX, destY).

get_path(*origX*, *origY*, *destX*, *destY*)
Get the shortest path from origXY to destXY.

Returns

Returns a list walking the path from orig to dest.

This excludes the starting point and includes the destination.

If no path is found then an empty list is returned.

Return type List[Tuple[int, int]]

class tdl.map.**Map**(*width*, *height*)
Field-of-view and path-finding on stored data.

Changed in version 4.1: *transparent*, *walkable*, and *fov* are now numpy boolean arrays.

Deprecated since version 3.2: *tcod.map.Map* should be used instead.

Set map conditions with the walkable and transparency attributes, this object can be iterated and checked for containment similar to consoles.

For example, you can set all tiles and transparent and walkable with the following code:

Example

```
>>> import tdl.map
>>> map_ = tdl.map.Map(80, 60)
>>> map_.transparent[:] = True
>>> map_.walkable[:] = True
```

transparent
Map transparency

Access this attribute with `map.transparent[x,y]`

Set to True to allow field-of-view rays, False will block field-of-view.

Transparent tiles only affect field-of-view.

walkable
Map accessibility

Access this attribute with `map.walkable[x,y]`

Set to True to allow path-finding through that tile, False will block passage to that tile.

Walkable tiles only affect path-finding.

fov
Map tiles touched by a field-of-view computation.

Access this attribute with `map.fov[x,y]`

Is True if a the tile is if view, otherwise False.

You can set to this attribute if you want, but you'll typically be using it to read the field-of-view of a *compute_fov* call.

compute_fov(*x*, *y*, *fov='PERMISSIVE'*, *radius=None*, *light_walls=True*, *sphere=True*, *cumulative=False*)

Compute the field-of-view of this Map and return an iterator of the points touched.

Parameters

- **x** (*int*) – Point of view, x-coordinate.
- **y** (*int*) – Point of view, y-coordinate.
- **fov** (*Text*) – The type of field-of-view to be used.

  Available types are: 'BASIC', 'DIAMOND', 'SHADOW', 'RESTRICTIVE', 'PERMISSIVE', 'PERMISSIVE0', 'PERMISSIVE1', ..., 'PERMISSIVE8'
- **radius** (*Optional[int]*) – Maximum view distance from the point of view.

  A value of 0 will give an infinite distance.
- **light_walls** (*bool*) – Light up walls, or only the floor.
- **sphere** (*bool*) – If True the lit area will be round instead of square.
- **cumulative** (*bool*) – If True the lit cells will accumulate instead of being cleared before the computation.

Returns

An iterator of (x, y) points of tiles touched by the field-of-view.

Return type Iterator[Tuple[int, int]]

compute_path(*start_x*, *start_y*, *dest_x*, *dest_y*, *diagonal_cost=1.4142135623730951*)

Get the shortest path between two points.

Parameters

- **start_x** (*int*) – Starting x-position.
- **start_y** (*int*) – Starting y-position.
- **dest_x** (*int*) – Destination x-position.
- **dest_y** (*int*) – Destination y-position.
- **diagonal_cost** (*float*) – Multiplier for diagonal movement.

  Can be set to zero to disable diagonal movement entirely.

Returns

The shortest list of points to the destination position from the starting position.

The start point is not included in this list.

Return type List[Tuple[int, int]]

tdl.map.**bresenham**(*x1*, *y1*, *x2*, *y2*)

Return a list of points in a bresenham line.

Implementation hastily copied from RogueBasin.

Returns

A list of (x, y) points, including both the start and end-points.

Return type List[Tuple[int, int]]

tdl.map.**quick_fov**(*x*, *y*, *callback*, *fov='PERMISSIVE'*, *radius=7.5*, *lightWalls=True*, *sphere=True*)

All field-of-view functionality in one call.

Before using this call be sure to make a function, lambda, or method that takes 2 positional parameters and returns True if light can pass through the tile or False for light-blocking tiles and for indexes that are out of bounds of the dungeon.

This function is 'quick' as in no hassle but can quickly become a very slow function call if a large radius is used or the callback provided itself isn't optimized.

Always check if the index is in bounds both in the callback and in the returned values. These values can go into the negatives as well.

Parameters

- **x** (*int*) – x center of the field-of-view
- **y** (*int*) – y center of the field-of-view
- **callback** (*Callable[[int, int], bool]*) – This should be a function that takes two positional arguments x,y and returns True if the tile at that position is transparent or False if the tile blocks light or is out of bounds.
- **fov** (*Text*) – The type of field-of-view to be used.

  Available types are: 'BASIC', 'DIAMOND', 'SHADOW', 'RESTRICTIVE', 'PERMISSIVE', 'PERMISSIVE0', 'PERMISSIVE1', ..., 'PERMISSIVE8'
- **radius** (*float*) – When sphere is True a floating point can be used to fine-tune the range. Otherwise the radius is just rounded up.

  Be careful as a large radius has an exponential affect on how long this function takes.
- **lightWalls** (*bool*) – Include or exclude wall tiles in the field-of-view.
- **sphere** (*bool*) – True for a spherical field-of-view. False for a square one.

Returns

A set of (x, y) points that are within the field-of-view.

Return type Set[Tuple[int, int]]

CHAPTER 20

tdl.noise

This module provides advanced noise generation.

Noise is sometimes used for over-world generation, height-maps, and cloud/mist/smoke effects among other things.

You can see examples of the available noise algorithms in the libtcod documentation here.

class `tdl.noise.`**`Noise`**(*algorithm='PERLIN'*, *mode='FLAT'*, *hurst=0.5*, *lacunarity=2.0*, *octaves=4.0*, *seed=None*, *dimensions=4*)

An advanced noise generator.

Deprecated since version 3.2: This class has been replaced by *tcod.noise.Noise*.

Parameters

- **`algorithm`** (*Text*) – The primary noise algorithm to be used.

  Can be one of 'PERLIN', 'SIMPLEX', 'WAVELET'

  - 'PERLIN' - A popular noise generator.
  - 'SIMPLEX' - In theory this is a slightly faster generator with less noticeable directional artifacts.
  - 'WAVELET' - A noise generator designed to reduce aliasing and not lose detail when summed into a fractal (as with the 'FBM' and 'TURBULENCE' modes.) This works faster at higher dimensions.

- **`mode`** (*Text*) – A secondary parameter to determine how noise is generated.

  Can be one of 'FLAT', 'FBM', 'TURBULENCE'

  - 'FLAT' - Generates the simplest form of noise. This mode does not use the hurst, lacunarity, and octaves parameters.
  - 'FBM' - Generates fractal brownian motion.
  - 'TURBULENCE' - Generates detailed noise with smoother and more natural transitions.

- **`hurst`** (*float*) – The hurst exponent.

This describes the raggedness of the resultant noise, with a higher value leading to a smoother noise. It should be in the 0.0-1.0 range.

This is only used in 'FBM' and 'TURBULENCE' modes.

- **lacunarity** (*float*) – A multiplier that determines how quickly the frequency increases for each successive octave.

  The frequency of each successive octave is equal to the product of the previous octave's frequency and the lacunarity value.

  This is only used in 'FBM' and 'TURBULENCE' modes.

- **octaves** (*float*) – Controls the amount of detail in the noise.

  This is only used in 'FBM' and 'TURBULENCE' modes.

- **seed** (*Hashable*) – You can use any hashable object to be a seed for the noise generator.

  If None is used then a random seed will be generated.

get_point(**position*)

Return the noise value of a specific position.

Example usage: value = noise.getPoint(x, y, z)

Parameters **position** (*Tuple[float, ..]*) – The point to sample at.

Returns

The noise value at position.

This will be a floating point in the 0.0-1.0 range.

Return type float

CHAPTER 21

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

L

M

N

P

Q

R

S

T

U

V

W

X

Y