python-tcod Documentation

Release 11.19.3

Kyle Stewart

Jan 22, 2021

Contents

Contents:

CHAPTER 1

Installation

Once installed, you'll be able to import the *tcod* and *libtcodpy* modules, as well as the deprecated *tdl* module.

Python 3.5 or above is required for a normal install. These instructions include installing Python if you don't have it yet.

There are known issues in very old versions of pip. If pip fails to install python-tcod then try updating it first.

1.1 Windows

First install a recent version of Python 3. Make sure Python is added to the Windows *PATH*.

If you don't already have it, then install the latest Microsoft Visual C++ Redistributable. **vc_redist.x86.exe** for a 32-bit install of Python, or **vc_redist.x64.exe** for a 64-bit install. You'll need to keep this in mind when distributing any libtcod program to end-users.

Then to install python-tcod run the following from a Windows command line:

```
py -m pip install tcod
```

If Python was installed for all users then you may need to add the `--user` flag to pip.

1.2 MacOS

The latest version of python-tcod only supports MacOS 10.9 (Mavericks) or later.

First install a recent version of Python 3.

Then to install using pip in a user environment, use the following command:

```
python3 -m pip install --user tcod
```

1.3 Linux (Debian-based)

On Linux python-tcod will need to be built from source. You can run this command to download python-tcod's dependencies with apt:

```
sudo apt install build-essential python3-dev python3-pip python3-numpy libsdl2-dev␣
↪libffi-dev libomp5
```

If your GCC version is less than 6.1, or your SDL version is less than 2.0.5, then you will need to perform a distribution upgrade before continuing.

Once dependences are resolved you can build and install python-tcod using pip in a user environment:

```
python3 -m pip install --user tcod
```

1.4 PyCharm

PyCharm will often run your project in a virtual environment, hiding any modules you installed system-wide. You must install python-tcod inside of the virtual environment in order for it to be importable in your projects scripts.

By default the bottom bar of PyCharm will have a tab labeled *terminal*. Open this tab and you should see a prompt with `(venv)` on it. This means your commands will run in the virtual environment of your project.

With a new project and virtual environment you should upgrade pip before installing python-tcod. You can do this by running the command:

```
python -m pip install --upgrade pip
```

If this for some reason failed, you may fall back on using *easy_install*:

```
easy_install --upgrade pip
```

After pip is upgraded you can install tcod with the following command:

```
pip install tcod
```

You can now use `import tcod`.

If you are working with multiple people or computers then it's recommend to pin the tcod version in a requirements.txt file. PyCharm will automatically update the virtual environment from these files.

1.5 Upgrading python-tcod

python-tcod is updated often, you can re-run pip with the `--upgrade` flag to ensure you have the latest version, for example:

```
python3 -m pip install --upgrade tcod
```

1.6 Upgrading from libtcodpy to python-tcod

libtcodpy is no longer maintained and using it can make it difficult to collaborate with developers across multiple operating systems, or to distribute to those platforms. New API features are only available on *python-tcod*.

You can recognize a libtcodpy program because it includes this file structure:

```
libtcodpy/ (or libtcodpy.py)
libtcod.dll (or libtcod-mingw.dll)
SDL2.dll (or SDL.dll)
```

First make sure your libtcodpy project works in Python 3. libtcodpy already supports both 2 and 3 so you don't need to worry about updating it, but you will need to worry about bit-size. If you're using a 32-bit version of Python 2 then you'll need to upgrade to a 32-bit version of Python 3 until libtcodpy can be completely removed.

For Python 3 you'll want the latest version of *tcod*, for Python 2 you'll need to install `tcod==6.0.7` instead, see the Python 2.7 instructions below.

Once you've installed python-tcod you can safely delete the `libtcodpy/` folder, the `libtcodpy.py` script, and all the DLL files of a libtcodpy program, python-tcod will seamlessly and immediately take the place of libtcodpy's API.

From then on anyone can follow the instructions in this guide to install python-tcod and your project will work for them regardless of their platform.

1.7 Distributing

Once your project is finished, it can be distributed using PyInstaller.

1.8 Python 2.7

While it's not recommended, you can still install *python-tcod* on *Python 2.7*.

Keep in mind that Python 2's end-of-life has already passed. You should not be starting any new projects in Python 2!

Follow the instructions for your platform normally. When it comes to install with pip, tell it to get python-tcod version 6:

```
python2 -m pip install tcod==6.0.7
```

CHAPTER 2

Glossary

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

These have been deprecated since version *8.5*.

libtcod-cffi This is the *cffi* implementation of libtcodpy, the original was made using *ctypes* which was more difficult to maintain.

libtcod-cffi has since been part of *python-tcod* providing all of the *libtcodpy* API until the newer features could be implemented.

python-tcod *python-tcod* is a superset of the *libtcodpy* API. The major additions include class functionality in returned objects, no manual memory management, pickle-able objects, and *numpy* array attributes in most objects.

The *numpy* functions in particular can be used to dramatically speed up the performance of a program compared to using *libtcodpy*.

python-tdl *tdl* is a high-level wrapper over *libtcodpy* although it now uses *python-tcod*, it doesn't do anything that you couldn't do yourself with just *libtcodpy* and Python.

It included a lot of core functions written in Python that most definitely shouldn't have been. *tdl* was very to use, but the cost was severe performance issues throughout the entire module. This left it impractical for any real use as a roguelike library.

Currently no new features are planned for *tdl*, instead new features are added to *libtcod* itself and then ported to *python-tcod*.

python-tdl and *libtcodpy* are included in installations of *python-tcod*.

libtcodpy *libtcodpy* is more or less a direct port of *libtcod*'s C API to Python. This caused a handful of issues including instances needing to be freed manually or else a memory leak would occur, and many functions performing badly in Python due to the need to call them frequently.

These issues are fixed in *python-tcod* which implements the full *libtcodpy* API. If *python-tcod* is installed then imports of *libtcodpy* are aliased to the *tcod* module. So if you come across a project using the original *libtcodpy* you can delete the *libtcodpy/* folder and then *python-tcod* will load instead.

CHAPTER 3

Changelog

Changes relevant to the users of python-tcod are documented here.

This project adheres to Semantic Versioning since v2.0.0

3.1 Unreleased

3.2 11.19.3 - 2021-01-07

Fixed

- Some wheels had broken version metadata.

3.3 11.19.2 - 2020-12-30

Changed

- Now bundles SDL 2.0.10 for MacOS and SDL 2.0.14 for Windows.

Fixed

- MacOS wheels were failing to bundle dependencies for SDL2.

3.4 11.19.1 - 2020-12-29

Fixed

- MacOS wheels failed to deploy for the previous version.

3.5 11.19.0 - 2020-12-29

Added

- Added the important *order* parameter to *Context.new_console*.

3.6 11.18.3 - 2020-12-28

Changed

- Now bundles SDL 2.0.14 for Windows/MacOS.

Deprecated

- Support for Python 3.5 will be dropped.
- *tcod.console_load_xp* has been deprecated, *tcod.console_from_xp* can load these files without modifying an existing console.

Fixed

- *tcod.console_from_xp* now has better error handling (instead of crashing.)
- Can now compile with SDL 2.0.14 headers.

3.7 11.18.2 - 2020-12-03

Fixed

- Fixed missing *tcod.FOV_SYMMETRIC_SHADOWCAST* constant.
- Fixed regression in *tcod.sys_get_current_resolution* behavior. This function now returns the monitor resolution as was previously expected.

3.8 11.18.1 - 2020-11-30

Fixed

- Code points from the Private Use Area will now print correctly.

3.9 11.18.0 - 2020-11-13

Added

- New context method *Context.new_console*.

Changed

- Using *libtcod 1.16.0-alpha.15*.

3.10 11.17.0 - 2020-10-30

Added

- New FOV implementation: *tcod.FOV_SYMMETRIC_SHADOWCAST*.

Changed

- Using *libtcod 1.16.0-alpha.14*.

3.11 11.16.1 - 2020-10-28

Deprecated

- Changed context deprecations to PendingDeprecationWarning to reduce mass panic from tutorial followers.

Fixed

- Fixed garbled titles and crashing on some platforms.

3.12 11.16.0 - 2020-10-23

Added

- Added *tcod.context.new* function.
- Contexts now support a CLI.
- You can now provide the window x,y position when making contexts.
- *tcod.noise.Noise* instances can now be indexed to generate noise maps.

Changed

- Using *libtcod 1.16.0-alpha.13*.
- The OpenGL 2 renderer can now use *SDL_HINT_RENDER_SCALE_QUALITY* to determine the tileset upscaling filter.
- Improved performance of the FOV_BASIC algorithm.

Deprecated

- *tcod.context.new_window* and *tcod.context.new_terminal* have been replaced by *tcod.context.new*.

Fixed

- Pathfinders will now work with boolean arrays.
- Console blits now ignore alpha compositing which would result in division by zero.
- *tcod.console_is_key_pressed* should work even if libtcod events are ignored.
- The *TCOD_RENDERER* and *TCOD_VSYNC* environment variables should work now.
- *FOV_PERMISSIVE* algorithm is now reentrant.

3.13 11.15.3 - 2020-07-30

Fixed

- *tcod.tileset.Tileset.remap*, codepoint and index were swapped.

3.14 11.15.2 - 2020-07-27

Fixed

- *tcod.path.dijkstra2d*, fixed corrupted output with int8 arrays.

3.15 11.15.1 - 2020-07-26

Changed

- *tcod.event.EventDispatch* now uses the absolute names for event type hints so that IDE's can better auto-complete method overrides.

Fixed

- Fixed libtcodpy heightmap data alignment issues on non-square maps.

3.16 11.15.0 - 2020-06-29

Added

- *tcod.path.SimpleGraph* for pathfinding on simple 2D arrays.

Changed

- *tcod.path.CustomGraph* now accepts an *order* parameter.

3.17 11.14.0 - 2020-06-23

Added

- New *tcod.los* module for NumPy-based line-of-sight algorithms. Includes *tcod.los.bresenham*.

Deprecated

- *tcod.line_where* and *tcod.line_iter* have been deprecated.

3.18 11.13.6 - 2020-06-19

Deprecated

- *console_init_root* and *console_set_custom_font* have been replaced by the modern API.
- All functions which handle SDL windows without a context are deprecated.
- All functions which modify a globally active tileset are deprecated.

- *tcod.map.Map* is deprecated, NumPy arrays should be passed to functions directly instead of through this class.

3.19 11.13.5 - 2020-06-15

Fixed

- Install requirements will no longer try to downgrade *cffi*.

3.20 11.13.4 - 2020-06-15

3.21 11.13.3 - 2020-06-13

Fixed

- *cffi* requirement has been updated to version *1.13.0*. The older versions raise TypeError's.

3.22 11.13.2 - 2020-06-12

Fixed

- SDL related errors during package installation are now more readable.

3.23 11.13.1 - 2020-05-30

Fixed

- *tcod.event.EventDispatch*: *ev_** methods now allow *Optional[T]* return types.

3.24 11.13.0 - 2020-05-22

Added

- *tcod.path*: New *Pathfinder* and *CustomGraph* classes.

Changed

- Added *edge_map* parameter to *tcod.path.dijkstra2d* and *tcod.path.hillclimb2d*.

Fixed

- tcod.console_init_root' and context initializing functions were not raising exceptions on failure.

3.25 11.12.1 - 2020-05-02

Fixed

- Prevent adding non-existent 2nd halves to potential double-wide charterers.

3.26 11.12.0 - 2020-04-30

Added

- Added *tcod.context* module. You now have more options for making libtcod controlled contexts.
- *tcod.tileset.load_tilesheet*: Load a simple tilesheet as a Tileset.
- *Tileset.remap*: Reassign codepoints to tiles on a Tileset.
- *tcod.tileset.CHARMAP_CP437*: Character mapping for *load_tilesheet*.
- *tcod.tileset.CHARMAP_TCOD*: Older libtcod layout.

Changed

- *EventDispatch.dispatch* can now return the values returned by the *ev_** methods. The class is now generic to support type checking these values.
- Event mouse coordinates are now strictly int types.
- Submodules are now implicitly imported.

3.27 11.11.4 - 2020-04-26

Changed

- Using *libtcod 1.16.0-alpha.10*.

Fixed

- Fixed characters being dropped when color codes were used.

3.28 11.11.3 - 2020-04-24

Changed

- Using *libtcod 1.16.0-alpha.9*.

Fixed

- *FOV_DIAMOND* and *FOV_RESTRICTIVE* algorithms are now reentrant. libtcod#48
- The *TCOD_VSYNC* environment variable was being ignored.

3.29 11.11.2 - 2020-04-22

3.30 11.11.1 - 2020-04-03

Changed

- Using *libtcod 1.16.0-alpha.8*.

Fixed

- Changing the active tileset now redraws tiles correctly on the next frame.

3.31 11.11.0 - 2020-04-02

Added

- Added *Console.close* as a more obvious way to close the active window of a root console.

Changed

- GCC is no longer needed to compile the library on Windows.
- Using *libtcod 1.16.0-alpha.7*.
- *tcod.console_flush* will now accept an RGB tuple as a *clear_color*.

Fixed

- Changing the active tileset will now properly show it on the next render.

3.32 11.10.0 - 2020-03-26

Added

- Added *tcod.tileset.load_bdf*, you can now load BDF fonts.
- *tcod.tileset.set_default* and *tcod.tileset.get_default* are now stable.

Changed

- Using *libtcod 1.16.0-alpha.6*.

Deprecated

- The *snap_to_integer* parameter in *tcod.console_flush* has been deprecated since it can cause minor scaling issues which don't exist when using *integer_scaling* instead.

3.33 11.9.2 - 2020-03-17

Fixed

- Fixed segfault after the Tileset returned by *tcod.tileset.get_default* goes out of scope.

3.34 11.9.1 - 2020-02-28

Changed

- Using *libtcod 1.16.0-alpha.5*.
- Mouse tile coordinates are now always zero before the first call to *tcod.console_flush*.

3.35 11.9.0 - 2020-02-22

Added

- New method *Tileset.render* renders an RGBA NumPy array from a tileset and a console.

3.36 11.8.2 - 2020-02-22

Fixed

- Prevent KeyError when representing unusual keyboard symbol constants.

3.37 11.8.1 - 2020-02-22

Changed

- Using *libtcod 1.16.0-alpha.4*.

Fixed

- Mouse tile coordinates are now correct on any resized window.

3.38 11.8.0 - 2020-02-21

Added

- Added *tcod.console.recommended_size* for when you want to change your main console size at runtime.
- Added *Console.tiles_rgb* as a replacement for *Console.tiles2*.

Changed

- Using *libtcod 1.16.0-alpha.3*.
- Added parameters to *tcod.console_flush*, you can now manually provide a console and adjust how it is presented.

Deprecated

- *Console.tiles2* is deprecated in favour of *Console.tiles_rgb*.
- *Console.buffer* is now deprecated in favour of *Console.tiles*, instead of the other way around.

Fixed

- Fixed keyboard state and mouse state functions losing state when events were flushed.

3.39 11.7.2 - 2020-02-16

Fixed

- Fixed regression in *tcod.console_clear*.

3.40 11.7.1 - 2020-02-16

Fixed

- Fixed regression in *Console.draw_frame*.
- The wavelet noise generator now excludes -1.0f and 1.0f as return values.
- Fixed console fading color regression.

3.41 11.7.0 - 2020-02-14

Changed

- Using *libtcod 1.16.0-alpha.2*.
- When a renderer fails to load it will now fallback to a different one. The order is: OPENGL2 -> OPENGL -> SDL2.
- The default renderer is now SDL2.
- The SDL and OPENGL renderers are no longer deprecated, but they now point to slightly different backward compatible implementations.

Deprecated

- The use of *libtcod.cfg* and *terminal.png* is deprecated.

Fixed

- *tcod.sys_update_char* now works with the newer renderers.
- Fixed buffer overflow in name generator.
- *tcod.image_from_console* now works with the newer renderers.
- New renderers now auto-load fonts from *libtcod.cfg* or *terminal.png*.

3.42 11.6.0 - 2019-12-05

Changed

- Console blit operations now perform per-cell alpha transparency.

3.43 11.5.1 - 2019-11-23

Fixed

- Python 3.8 wheels failed to deploy.

3.44 11.5.0 - 2019-11-22

Changed

- Quarter block elements are now rendered using Unicode instead of a custom encoding.

Fixed

- *OPENGL* and *GLSL* renderers were not properly clearing space characters.

3.45 11.4.1 - 2019-10-15

Added

- Uploaded Python 3.8 wheels to PyPI.

3.46 11.4.0 - 2019-09-20

Added

- Added *__array_interface__* to the Image class.
- Added *Console.draw_semigraphics* as a replacement for blit_2x functions. *draw_semigraphics* can handle array-like objects.
- *Image.from_array* class method creates an Image from an array-like object.
- *tcod.image.load* loads a PNG file as an RGBA array.

Changed

- *Console.tiles* is now named *Console.buffer*.

3.47 11.3.0 - 2019-09-06

Added

- New attribute *Console.tiles2* is similar to *Console.tiles* but without an alpha channel.

3.48 11.2.2 - 2019-08-25

Fixed

- Fixed a regression preventing PyInstaller distributions from loading SDL2.

3.49 11.2.1 - 2019-08-25

3.50 11.2.0 - 2019-08-24

Added

- *tcod.path.dijkstra2d*: Computes Dijkstra from an arbitrary initial state.
- *tcod.path.hillclimb2d*: Returns a path from a distance array.
- *tcod.path.maxarray*: Creates arrays filled with maximum finite values.

Fixed

- Changing the tiles of an active tileset on OPENGL2 will no longer leave temporary artifact tiles.
- It's now harder to accidentally import tcod's internal modules.

3.51 11.1.2 - 2019-08-02

Changed

- Now bundles SDL 2.0.10 for Windows/MacOS.

Fixed

- Can now parse SDL 2.0.10 headers during installation without crashing.

3.52 11.1.1 - 2019-08-01

Deprecated

- Using an out-of-bounds index for field-of-view operations now raises a warning, which will later become an error.

Fixed

- Changing the tiles of an active tileset will now work correctly.

3.53 11.1.0 - 2019-07-05

Added

- You can now set the *TCOD_RENDERER* and *TCOD_VSYNC* environment variables to force specific options to be used. Example: `TCOD_RENDERER=sdl2 TCOD_VSYNC=1`

Changed

- *tcod.sys_set_renderer* now raises an exception if it fails.

Fixed

- *tcod.console_map_ascii_code_to_font* functions will now work when called before *tcod.console_init_root*.

3.54 11.0.2 - 2019-06-21

Changed

- You no longer need OpenGL to build python-tcod.

3.55 11.0.1 - 2019-06-21

Changed

- Better runtime checks for Windows dependencies should now give distinct errors depending on if the issue is SDL2 or missing redistributables.

Fixed

- Changed NumPy type hints from *np.array* to *np.ndarray* which should resolve issues.

3.56 11.0.0 - 2019-06-14

Changed

- *tcod.map.compute_fov* now takes a 2-item tuple instead of separate *x* and *y* parameters. This causes less confusion over how axes are aligned.

3.57 10.1.1 - 2019-06-02

Changed

- Better string representations for *tcod.event.Event* subclasses.

Fixed

- Fixed regressions in text alignment for non-rectangle print functions.

3.58 10.1.0 - 2019-05-24

Added

- *tcod.console_init_root* now has an optional *vsync* parameter.

3.59 10.0.5 - 2019-05-17

Fixed

- Fixed shader compilation issues in the OPENGL2 renderer.
- Fallback fonts should fail less on Linux.

3.60 10.0.4 - 2019-05-17

Changed

- Now depends on cffi 0.12 or later.

Fixed

- *tcod.console_init_root* and *tcod.console_set_custom_font* will raise exceptions instead of terminating.
- Fixed issues preventing *tcod.event* from working on 32-bit Windows.

3.61 10.0.3 - 2019-05-10

Fixed

- Corrected bounding box issues with the *Console.print_box* method.

3.62 10.0.2 - 2019-04-26

Fixed

- Resolved Color warnings when importing tcod.
- When compiling, fixed a name conflict with endianness macros on FreeBSD.

3.63 10.0.1 - 2019-04-19

Fixed

- Fixed horizontal alignment for TrueType fonts.
- Fixed taking screenshots with the older SDL renderer.

3.64 10.0.0 - 2019-03-29

Added

- New *Console.tiles* array attribute.

Changed

- *Console.DTYPE* changed to add alpha to its color types.

Fixed

- Console printing was ignoring color codes at the beginning of a string.

3.65 9.3.0 - 2019-03-15

Added

- The SDL2/OPENGL2 renderers can potentially use a fall-back font when none are provided.
- New function *tcod.event.get_mouse_state*.
- New function *tcod.map.compute_fov* lets you get a visibility array directly from a transparency array.

Deprecated

- The following functions and classes have been deprecated. - *tcod.Key* - *tcod.Mouse* - *tcod.mouse_get_status* - *tcod.console_is_window_closed* - *tcod.console_check_for_keypress* - *tcod.console_wait_for_keypress* - *tcod.console_delete* - *tcod.sys_check_for_event* - *tcod.sys_wait_for_event*
- The SDL, OPENGL, and GLSL renderers have been deprecated.
- Many libtcodpy functions have been marked with PendingDeprecationWarning's.

Fixed

- To be more compatible with libtcodpy *tcod.console_init_root* will default to the SDL render, but will raise warnings when an old renderer is used.

3.66 9.2.5 - 2019-03-04

Fixed

- Fixed *tcod.namegen_generate_custom*.

3.67 9.2.4 - 2019-03-02

Fixed

- The *tcod* package is has been marked as typed and will now work with MyPy.

3.68 9.2.3 - 2019-03-01

Deprecated

- The behavior for negative indexes on the new print functions may change in the future.
- Methods and functionality preventing *tcod.Color* from behaving like a tuple have been deprecated.

3.69 9.2.2 - 2019-02-26

Fixed

- *Console.print_box* wasn't setting the background color by default.

3.70 9.2.1 - 2019-02-25

Fixed

- *tcod.sys_get_char_size* fixed on the new renderers.

3.71 9.2.0 - 2019-02-24

Added

- New *tcod.console.get_height_rect* function, which can be used to get the height of a print call without an existing console.
- New *tcod.tileset* module, with a *set_truetype_font* function.

Fixed

- The new print methods now handle alignment according to how they were documented.
- *SDL2* and *OPENGL2* now support screenshots.
- Windows and MacOS builds now restrict exported SDL2 symbols to only SDL 2.0.5; This will avoid hard to debug import errors when the wrong version of SDL is dynamically linked.
- The root console now starts with a white foreground.

3.72 9.1.0 - 2019-02-23

Added

- Added the *tcod.random.MULTIPLY_WITH_CARRY* constant.

Changed

- The overhead for warnings has been reduced when running Python with the optimize *-O* flag.
- *tcod.random.Random* now provides a default algorithm.

3.73 9.0.0 - 2019-02-17

Changed

- New console methods now default to an *fg* and *bg* of None instead of white-on-black.

3.74 8.5.0 - 2019-02-15

Added

- *tcod.console.Console* now supports *str* and *repr*.
- Added new Console methods which are independent from the console defaults.
- You can now give an array when initializing a *tcod.console.Console* instance.
- *Console.clear* can now take *ch*, *fg*, and *bg* parameters.

Changed

- Updated libtcod to 1.10.6
- Printing generates more compact layouts.

Deprecated

- Most libtcodpy console functions have been replaced by the tcod.console module.
- Deprecated the *set_key_color* functions. You can pass key colors to *Console.blit* instead.
- *Console.clear* should be given the colors to clear with as parameters, rather than by using *default_fg* or *default_bg*.
- Most functions which depend on console default values have been deprecated. The new deprecation warnings will give details on how to make default values explicit.

Fixed

- *tcod.console.Console.blit* was ignoring the key color set by *Console.set_key_color*.
- The *SDL2* and *OPENGL2* renders can now large numbers of tiles.

3.75 8.4.3 - 2019-02-06

Changed

- Updated libtcod to 1.10.5
- The SDL2/OPENGL2 renderers will now auto-detect a custom fonts key-color.

3.76 8.4.2 - 2019-02-05

Deprecated

- The tdl module has been deprecated.
- The libtcodpy parser functions have been deprecated.

Fixed

- *tcod.image_is_pixel_transparent* and *tcod.image_get_alpha* now return values.
- *Console.print_frame* was clearing tiles outside if its bounds.
- The *FONT_LAYOUT_CP437* layout was incorrect.

3.77 8.4.1 - 2019-02-01

Fixed

- Window event types were not upper-case.
- Fixed regression where libtcodpy mouse wheel events unset mouse coordinates.

3.78 8.4.0 - 2019-01-31

Added

- Added tcod.event module, based off of the sdlevent.py shim.

Changed

- Updated libtcod to 1.10.3

Fixed

- Fixed libtcodpy *struct_add_value_list* function.
- Use correct math for tile-based delta in mouse events.
- New renderers now support tile-based mouse coordinates.
- SDL2 renderer will now properly refresh after the window is resized.

3.79 8.3.2 - 2018-12-28

Fixed

- Fixed rare access violations for some functions which took strings as parameters, such as *tcod.console_init_root*.

3.80 8.3.1 - 2018-12-28

Fixed

- libtcodpy key and mouse functions will no longer accept the wrong types.
- The *new_struct* method was not being called for libtcodpy's custom parsers.

3.81 8.3.0 - 2018-12-08

Added

- Added BSP traversal methods in tcod.bsp for parity with libtcodpy.

Deprecated

- Already deprecated bsp functions are now even more deprecated.

3.82 8.2.0 - 2018-11-27

Added

- New layout *tcod.FONT_LAYOUT_CP437*.

Changed

- Updated libtcod to 1.10.2
- *tcod.console_print_frame* and *Console.print_frame* now support Unicode strings.

Deprecated

- Deprecated using bytes strings for all printing functions.

Fixed

- Console objects are now initialized with spaces. This fixes some blit operations.
- Unicode code-points above U+FFFF will now work on all platforms.

3.83 8.1.1 - 2018-11-16

Fixed

- Printing a frame with an empty string no longer displays a title bar.

3.84 8.1.0 - 2018-11-15

Changed

- Heightmap functions now support 'F_CONTIGUOUS' arrays.
- *tcod.heightmap_new* now has an *order* parameter.
- Updated SDL to 2.0.9

Deprecated

- Deprecated heightmap functions which sample noise grids, this can be done using the *Noise.sample_ogrid* method.

3.85 8.0.0 - 2018-11-02

Changed

- The default renderer can now be anything if not set manually.
- Better error message for when a font file isn't found.

3.86 7.0.1 - 2018-10-27

Fixed

- Building from source was failing because *console_2tris.glsl** was missing from source distributions.

3.87 7.0.0 - 2018-10-25

Added

- New *RENDERER_SDL2* and *RENDERER_OPENGL2* renderers.

Changed

- Updated libtcod to 1.9.0
- Now requires SDL 2.0.5, which is not trivially installable on Ubuntu 16.04 LTS.

Removed

- Dropped support for Python versions before 3.5
- Dropped support for MacOS versions before 10.9 Mavericks.

3.88 6.0.7 - 2018-10-24

Fixed

- The root console no longer loses track of buffers and console defaults on a renderer change.

3.89 6.0.6 - 2018-10-01

Fixed

- Replaced missing wheels for older and 32-bit versions of MacOS.

3.90 6.0.5 - 2018-09-28

Fixed

- Resolved CDefError error during source installs.

3.91 6.0.4 - 2018-09-11

Fixed

- tcod.Key right-hand modifiers are now set independently at initialization, instead of mirroring the left-hand modifier value.

3.92 6.0.3 - 2018-09-05

Fixed

- tcod.Key and tcod.Mouse no longer ignore initiation parameters.

3.93 6.0.2 - 2018-08-28

Fixed

- Fixed color constants missing at build-time.

3.94 6.0.1 - 2018-08-24

Fixed

- Source distributions were missing C++ source files.

3.95 6.0.0 - 2018-08-23

Changed

- Project renamed to tcod on PyPI.

Deprecated

- Passing bytes strings to libtcodpy print functions is deprecated.

Fixed

- Fixed libtcodpy print functions not accepting bytes strings.
- libtcod constants are now generated at build-time fixing static analysis tools.

3.96 5.0.1 - 2018-07-08

Fixed

- tdl.event no longer crashes with StopIteration on Python 3.7

3.97 5.0.0 - 2018-07-05

Changed

- tcod.path: all classes now use *shape* instead of *width* and *height*.
- tcod.path now respects NumPy array shape, instead of assuming that arrays need to be transposed from C memory order. From now on *x* and *y* mean 1st and 2nd axis. This doesn't affect non-NumPy code.
- tcod.path now has full support of non-contiguous memory.

3.98 4.6.1 - 2018-06-30

Added

- New function *tcod.line_where* for indexing NumPy arrays using a Bresenham line.

Deprecated

- Python 2.7 support will be dropped in the near future.

3.99 4.5.2 - 2018-06-29

Added

- New wheels for Python3.7 on Windows.

Fixed

- Arrays from *tcod.heightmap_new* are now properly zeroed out.

3.100 4.5.1 - 2018-06-23

Deprecated

- Deprecated all libtcodpy map functions.

Fixed

- *tcod.map_copy* could break the *tcod.map.Map* class.
- *tcod.map_clear transparent* and *walkable* parameters were reversed.
- When multiple SDL2 headers were installed, the wrong ones would be used when the library is built.
- Fails to build via pip unless Numpy is installed first.

3.101 4.5.0 - 2018-06-12

Changed

- Updated libtcod to v1.7.0
- Updated SDL to v2.0.8
- Error messages when failing to create an SDL window should be a less vague.
- You no longer need to initialize libtcod before you can print to an off-screen console.

Fixed

- Avoid crashes if the root console has a character code higher than expected.

Removed

- No more debug output when loading fonts.

3.102 4.4.0 - 2018-05-02

Added

- Added the libtcodpy module as an alias for tcod. Actual use of it is deprecated, it exists primarily for backward compatibility.
- Adding missing libtcodpy functions *console_has_mouse_focus* and *console_is_active*.

Changed

- Updated libtcod to v1.6.6

3.103 4.3.2 - 2018-03-18

Deprecated

- Deprecated the use of falsy console parameters with libtcodpy functions.

Fixed

- Fixed libtcodpy image functions not supporting falsy console parameters.
- Fixed tdl *Window.get_char* method. (Kaczor2704)

3.104 4.3.1 - 2018-03-07

Fixed

- Fixed cffi.api.FFIError "unsupported expression: expected a simple numeric constant" error when building on platforms with an older cffi module and newer SDL headers.
- tcod/tdl Map and Console objects were not saving stride data when pickled.

3.105 4.3.0 - 2018-02-01

Added

- You can now set the numpy memory order on tcod.console.Console, tcod.map.Map, and tdl.map.Map objects well as from the tcod.console_init_root function.

Changed

- The *console_init_root title* parameter is now optional.

Fixed

- OpenGL renderer alpha blending is now consistent with all other render modes.

3.106 4.2.3 - 2018-01-06

Fixed

- Fixed setup.py regression that could prevent building outside of the git repository.

3.107 4.2.2 - 2018-01-06

Fixed

- The Windows dynamic linker will now prefer the bundled version of SDL. This fixes: "ImportError: DLL load failed: The specified procedure could not be found."
- *key.c* is no longer set when *key.vk == KEY_TEXT*, this fixes a regression which was causing events to be heard twice in the libtcod/Python tutorial.

3.108 4.2.0 - 2018-01-02

Changed

- Updated libtcod backend to v1.6.4
- Updated SDL to v2.0.7 for Windows/MacOS.

Removed

- Source distributions no longer include tests, examples, or fonts. Find these on GitHub.

Fixed

- Fixed "final link failed: Nonrepresentable section on output" error when compiling for Linux.
- *tcod.console_init_root* defaults to the SDL renderer, other renderers cause issues with mouse movement events.

3.109 4.1.1 - 2017-11-02

Fixed

- Fixed *ConsoleBuffer.blit* regression.

- Console defaults corrected, the root console's blend mode and alignment is the default value for newly made Console's.
- You can give a byte string as a filename to load parsers.

3.110 4.1.0 - 2017-07-19

Added

- tdl Map class can now be pickled.

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.

3.111 4.0.1 - 2017-07-12

Fixed

- tdl: Fixed NameError in *set_fps*.

3.112 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.

3.113 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.

3.114 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.

3.115 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.

3.116 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.

3.117 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.

3.118 2.0.1 - 2017-02-22

Fixed

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

3.119 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.

3.120 1.6.0 - 2016-11-18

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

3.121 1.5.3 - 2016-06-04

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

3.122 1.5.2 - 2016-03-11

- Fixed non-square Map instances

3.123 1.5.1 - 2015-12-20

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

3.124 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

3.125 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.

3.126 1.3.1 - 2015-06-19

- Fixed pathfinding regressions.

3.127 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.

3.128 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.

3.129 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

3.130 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.

3.131 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.

3.132 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.

3.133 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.

3.134 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.

3.135 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.

3.136 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

3.137 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.

3.138 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.

3.139 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.

3.140 1.0.0 - 2009-01-31

- First public release.

CHAPTER 4

Getting Started

Python 3 and python-tcod must be installed, see *Installation*.

4.1 Fixed-size console

This example is a hello world script which handles font loading, fixed-sized consoles, window contexts, and event handling. This example requires the dejavu10x10_gs_tc.png font to be in the same directory as the script.

By default this will create a window which can be resized and the fixed-size console will be stretched to fit the window. You can add arguments to `Context.present` to fix the aspect ratio or only scale the console by integer increments.

Example:

```
#!/usr/bin/env python3
# Make sure 'dejavu10x10_gs_tc.png' is in the same directory as this script.
import tcod

WIDTH, HEIGHT = 80, 60  # Console width and height in tiles.


def main() -> None:
    """Script entry point."""
    # Load the font, a 32 by 8 tile font with libtcod's old character layout.
    tileset = tcod.tileset.load_tilesheet(
        "dejavu10x10_gs_tc.png", 32, 8, tcod.tileset.CHARMAP_TCOD,
    )
    # Create the main console.
    console = tcod.Console(WIDTH, HEIGHT, order="F")
    # Create a window based on this console and tileset.
    with tcod.context.new(  # New window for a console of size columns×rows.
        columns=console.width, rows=console.height, tileset=tileset,
    ) as context:
        while True:  # Main loop, runs until SystemExit is raised.
            console.clear()
```

(continues on next page)

(continued from previous page)

```
            console.print(x=0, y=0, string="Hello World!")
            context.present(console)  # Show the console.

            for event in tcod.event.wait():
                context.convert_event(event)  # Sets tile coordinates for mouse␣
↪events.
                print(event)  # Print event information to stdout.
                if event.type == "QUIT":
                    raise SystemExit()
        # The window will be closed after the above with-block exits.


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

4.2 Dynamically-sized console

The next example shows a more advanced setup. A maximized window is created and the console is dynamically scaled to fit within it. If the window is resized then the console will be resized to match it.

Because a tileset wasn't manually loaded in this example an OS dependent fallback font will be used. This is useful for prototyping but it's not recommended to release with this font since it can fail to load on some platforms.

The *integer_scaling* parameter to `Context.present` prevents the console from being slightly stretched, since the console will rarely be the prefect size a small border will exist. This border is black by default but can be changed to another color.

You'll need to consider things like the console being too small for your code to handle or the tiles being small compared to an extra large monitor resolution. `Context.new_console` can be given a minimum size that it will never go below.

You can call `Context.new_console` every frame or only when the window is resized. This example creates a new console every frame instead of clearing the console every frame and replacing it only on resizing the window.

Example:

```
#!/usr/bin/env python3
import tcod

WIDTH, HEIGHT = 720, 480  # Window pixel resolution (when not maximized.)
FLAGS = tcod.context.SDL_WINDOW_RESIZABLE | tcod.context.SDL_WINDOW_MAXIMIZED


def main() -> None:
    """Script entry point."""
    with tcod.context.new(  # New window with pixel resolution of width×height.
        width=WIDTH, height=HEIGHT, sdl_window_flags=FLAGS
    ) as context:
        while True:
            console = context.new_console(order="F")
            console.print(0, 0, "Hello World")
            context.present(console, integer_scaling=True)

            for event in tcod.event.wait():
                context.convert_event(event)  # Sets tile coordinates for mouse␣
↪events.
```

(continues on next page)

(continued from previous page)

```
                print(event)
                if event.type == "QUIT":
                    raise SystemExit()
                if event.type == "WINDOWRESIZED":
                    pass  # The next call to context.new_console may return a␣
↪different size.


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

CHAPTER 5

Character Table Reference

This document exists as an easy reference for using non-ASCII glyphs in standard tcod functions.

Tile Index is the position of the glyph in the tileset image. This is relevant for loading the tileset and for using `Tileset.remap` to reassign tiles to new code points.

Unicode is the Unicode code point as a hexadecimal number. You can use `chr` to convert these numbers into a string. Character maps such as `tcod.tileset.CHARMAP_CP437` are simply a list of Unicode numbers, where the index of the list is the *Tile Index*.

String is the Python string for that character. This lets you use that character inline with print functions. These will work with `ord` to convert them into a number.

Name is the official name of a Unicode character.

The symbols currently shown under *String* are provided by your browser, they typically won't match the graphics provided by your tileset or could even be missing from your browsers font entirely. You could experience similar issues with your editor and IDE.

5.1 Code Page 437

The layout for tilesets loaded with: `tcod.tileset.CHARMAP_CP437`

This is one of the more common character mappings. Used for several games in the DOS era, and still used today whenever you want an old school aesthetic.

The Dwarf Fortress community is known to have a large collection of tilesets in this format: https://dwarffortresswiki.org/index.php/Tileset_repository

Wikipedia also has a good reference for this character mapping: https://en.wikipedia.org/wiki/Code_page_437

Tile Index	Unicode	String	Name
0	0x00	'\x00'	
1	0x263A	''	WHITE SMILING FACE

Continued on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
2	0x263B	‘’	BLACK SMILING FACE
3	0x2665	‘’	BLACK HEART SUIT
4	0x2666	‘’	BLACK DIAMOND SUIT
5	0x2663	‘’	BLACK CLUB SUIT
6	0x2660	‘’	BLACK SPADE SUIT
7	0x2022	‘•’	BULLET
8	0x25D8	‘’	INVERSE BULLET
9	0x25CB	‘’	WHITE CIRCLE
10	0x25D9	‘’	INVERSE WHITE CIRCLE
11	0x2642	‘’	MALE SIGN
12	0x2640	‘’	FEMALE SIGN
13	0x266A	‘♪’	EIGHTH NOTE
14	0x266B	‘’	BEAMED EIGHTH NOTES
15	0x263C	‘’	WHITE SUN WITH RAYS
16	0x25BA	‘’	BLACK RIGHT-POINTING POINTER
17	0x25C4	‘’	BLACK LEFT-POINTING POINTER
18	0x2195	‘’	UP DOWN ARROW
19	0x203C	‘’	DOUBLE EXCLAMATION MARK
20	0xB6	‘¶’	PILCROW SIGN
21	0xA7	‘§’	SECTION SIGN
22	0x25AC	‘’	BLACK RECTANGLE
23	0x21A8	‘’	UP DOWN ARROW WITH BASE
24	0x2191	‘↑’	UPWARDS ARROW
25	0x2193	‘↓’	DOWNWARDS ARROW
26	0x2192	‘→’	RIGHTWARDS ARROW
27	0x2190	‘←’	LEFTWARDS ARROW
28	0x221F	‘’	RIGHT ANGLE
29	0x2194	‘’	LEFT RIGHT ARROW
30	0x25B2	‘’	BLACK UP-POINTING TRIANGLE
31	0x25BC	‘’	BLACK DOWN-POINTING TRIANGLE
32	0x20	‘ ‘	SPACE
33	0x21	‘!’	EXCLAMATION MARK
34	0x22	‘”’	QUOTATION MARK
35	0x23	‘#’	NUMBER SIGN
36	0x24	‘$’	DOLLAR SIGN
37	0x25	‘%’	PERCENT SIGN
38	0x26	‘&’	AMPERSAND
39	0x27	“’”	APOSTROPHE
40	0x28	‘(‘	LEFT PARENTHESIS
41	0x29	‘)’	RIGHT PARENTHESIS
42	0x2A	‘*’	ASTERISK
43	0x2B	‘+’	PLUS SIGN
44	0x2C	‘,’	COMMA
45	0x2D	‘-‘	HYPHEN-MINUS
46	0x2E	‘.’	FULL STOP
47	0x2F	‘/’	SOLIDUS
48	0x30	‘0’	DIGIT ZERO
49	0x31	‘1’	DIGIT ONE
50	0x32	‘2’	DIGIT TWO

Continued on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
51	0x33	‘3’	DIGIT THREE
52	0x34	‘4’	DIGIT FOUR
53	0x35	‘5’	DIGIT FIVE
54	0x36	‘6’	DIGIT SIX
55	0x37	‘7’	DIGIT SEVEN
56	0x38	‘8’	DIGIT EIGHT
57	0x39	‘9’	DIGIT NINE
58	0x3A	‘:’	COLON
59	0x3B	‘;’	SEMICOLON
60	0x3C	‘<’	LESS-THAN SIGN
61	0x3D	‘=’	EQUALS SIGN
62	0x3E	‘>’	GREATER-THAN SIGN
63	0x3F	‘?’	QUESTION MARK
64	0x40	‘@’	COMMERCIAL AT
65	0x41	‘A’	LATIN CAPITAL LETTER A
66	0x42	‘B’	LATIN CAPITAL LETTER B
67	0x43	‘C’	LATIN CAPITAL LETTER C
68	0x44	‘D’	LATIN CAPITAL LETTER D
69	0x45	‘E’	LATIN CAPITAL LETTER E
70	0x46	‘F’	LATIN CAPITAL LETTER F
71	0x47	‘G’	LATIN CAPITAL LETTER G
72	0x48	‘H’	LATIN CAPITAL LETTER H
73	0x49	‘I’	LATIN CAPITAL LETTER I
74	0x4A	‘J’	LATIN CAPITAL LETTER J
75	0x4B	‘K’	LATIN CAPITAL LETTER K
76	0x4C	‘L’	LATIN CAPITAL LETTER L
77	0x4D	‘M’	LATIN CAPITAL LETTER M
78	0x4E	‘N’	LATIN CAPITAL LETTER N
79	0x4F	‘O’	LATIN CAPITAL LETTER O
80	0x50	‘P’	LATIN CAPITAL LETTER P
81	0x51	‘Q’	LATIN CAPITAL LETTER Q
82	0x52	‘R’	LATIN CAPITAL LETTER R
83	0x53	‘S’	LATIN CAPITAL LETTER S
84	0x54	‘T’	LATIN CAPITAL LETTER T
85	0x55	‘U’	LATIN CAPITAL LETTER U
86	0x56	‘V’	LATIN CAPITAL LETTER V
87	0x57	‘W’	LATIN CAPITAL LETTER W
88	0x58	‘X’	LATIN CAPITAL LETTER X
89	0x59	‘Y’	LATIN CAPITAL LETTER Y
90	0x5A	‘Z’	LATIN CAPITAL LETTER Z
91	0x5B	‘[‘	LEFT SQUARE BRACKET
92	0x5C	‘\’	REVERSE SOLIDUS
93	0x5D	‘]’	RIGHT SQUARE BRACKET
94	0x5E	‘^’	CIRCUMFLEX ACCENT
95	0x5F	‘_’	LOW LINE
96	0x60	‘‘’	GRAVE ACCENT
97	0x61	‘a’	LATIN SMALL LETTER A
98	0x62	‘b’	LATIN SMALL LETTER B
99	0x63	‘c’	LATIN SMALL LETTER C

Continued on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
100	0x64	‘d’	LATIN SMALL LETTER D
101	0x65	‘e’	LATIN SMALL LETTER E
102	0x66	‘f’	LATIN SMALL LETTER F
103	0x67	‘g’	LATIN SMALL LETTER G
104	0x68	‘h’	LATIN SMALL LETTER H
105	0x69	‘i’	LATIN SMALL LETTER I
106	0x6A	‘j’	LATIN SMALL LETTER J
107	0x6B	‘k’	LATIN SMALL LETTER K
108	0x6C	‘l’	LATIN SMALL LETTER L
109	0x6D	‘m’	LATIN SMALL LETTER M
110	0x6E	‘n’	LATIN SMALL LETTER N
111	0x6F	‘o’	LATIN SMALL LETTER O
112	0x70	‘p’	LATIN SMALL LETTER P
113	0x71	‘q’	LATIN SMALL LETTER Q
114	0x72	‘r’	LATIN SMALL LETTER R
115	0x73	‘s’	LATIN SMALL LETTER S
116	0x74	‘t’	LATIN SMALL LETTER T
117	0x75	‘u’	LATIN SMALL LETTER U
118	0x76	‘v’	LATIN SMALL LETTER V
119	0x77	‘w’	LATIN SMALL LETTER W
120	0x78	‘x’	LATIN SMALL LETTER X
121	0x79	‘y’	LATIN SMALL LETTER Y
122	0x7A	‘z’	LATIN SMALL LETTER Z
123	0x7B	‘{‘	LEFT CURLY BRACKET
124	0x7C	‘|’	VERTICAL LINE
125	0x7D	‘}’	RIGHT CURLY BRACKET
126	0x7E	‘~’	TILDE
127	0x7F	‘\x7f’	
128	0xC7	‘Ç’	LATIN CAPITAL LETTER C WITH CEDILLA
129	0xFC	‘ü’	LATIN SMALL LETTER U WITH DIAERESIS
130	0xE9	‘é’	LATIN SMALL LETTER E WITH ACUTE
131	0xE2	‘â’	LATIN SMALL LETTER A WITH CIRCUMFLEX
132	0xE4	‘ä’	LATIN SMALL LETTER A WITH DIAERESIS
133	0xE0	‘à’	LATIN SMALL LETTER A WITH GRAVE
134	0xE5	‘å’	LATIN SMALL LETTER A WITH RING ABOVE
135	0xE7	‘ç’	LATIN SMALL LETTER C WITH CEDILLA
136	0xEA	‘ê’	LATIN SMALL LETTER E WITH CIRCUMFLEX
137	0xEB	‘ë’	LATIN SMALL LETTER E WITH DIAERESIS
138	0xE8	‘è’	LATIN SMALL LETTER E WITH GRAVE
139	0xEF	‘ï’	LATIN SMALL LETTER I WITH DIAERESIS
140	0xEE	‘î’	LATIN SMALL LETTER I WITH CIRCUMFLEX
141	0xEC	‘ì’	LATIN SMALL LETTER I WITH GRAVE
142	0xC4	‘Ä’	LATIN CAPITAL LETTER A WITH DIAERESIS
143	0xC5	‘Å’	LATIN CAPITAL LETTER A WITH RING ABOVE
144	0xC9	‘É’	LATIN CAPITAL LETTER E WITH ACUTE
145	0xE6	‘æ’	LATIN SMALL LETTER AE
146	0xC6	‘Æ’	LATIN CAPITAL LETTER AE
147	0xF4	‘ô’	LATIN SMALL LETTER O WITH CIRCUMFLEX
148	0xF6	‘ö’	LATIN SMALL LETTER O WITH DIAERESIS

Continued on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
149	0xF2	‘ò’	LATIN SMALL LETTER O WITH GRAVE
150	0xFB	‘û’	LATIN SMALL LETTER U WITH CIRCUMFLEX
151	0xF9	‘ù’	LATIN SMALL LETTER U WITH GRAVE
152	0xFF	‘ÿ’	LATIN SMALL LETTER Y WITH DIAERESIS
153	0xD6	‘Ö’	LATIN CAPITAL LETTER O WITH DIAERESIS
154	0xDC	‘Ü’	LATIN CAPITAL LETTER U WITH DIAERESIS
155	0xA2	‘¢’	CENT SIGN
156	0xA3	‘£’	POUND SIGN
157	0xA5	‘¥’	YEN SIGN
158	0x20A7	‘’	PESETA SIGN
159	0x0192	‘ƒ’	LATIN SMALL LETTER F WITH HOOK
160	0xE1	‘á’	LATIN SMALL LETTER A WITH ACUTE
161	0xED	‘í’	LATIN SMALL LETTER I WITH ACUTE
162	0xF3	‘ó’	LATIN SMALL LETTER O WITH ACUTE
163	0xFA	‘ú’	LATIN SMALL LETTER U WITH ACUTE
164	0xF1	‘ñ’	LATIN SMALL LETTER N WITH TILDE
165	0xD1	‘Ñ’	LATIN CAPITAL LETTER N WITH TILDE
166	0xAA	‘ª’	FEMININE ORDINAL INDICATOR
167	0xBA	‘º’	MASCULINE ORDINAL INDICATOR
168	0xBF	‘¿’	INVERTED QUESTION MARK
169	0x2310	‘’	REVERSED NOT SIGN
170	0xAC	‘¬’	NOT SIGN
171	0xBD	‘½’	VULGAR FRACTION ONE HALF
172	0xBC	‘¼’	VULGAR FRACTION ONE QUARTER
173	0xA1	‘¡’	INVERTED EXCLAMATION MARK
174	0xAB	‘«’	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
175	0xBB	‘»’	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
176	0x2591	‘’	LIGHT SHADE
177	0x2592	‘’	MEDIUM SHADE
178	0x2593	‘’	DARK SHADE
179	0x2502	‘ ’	BOX DRAWINGS LIGHT VERTICAL
180	0x2524	‘’	BOX DRAWINGS LIGHT VERTICAL AND LEFT
181	0x2561	‘’	BOX DRAWINGS VERTICAL SINGLE AND LEFT DOUBLE
182	0x2562	‘’	BOX DRAWINGS VERTICAL DOUBLE AND LEFT SINGLE
183	0x2556	‘’	BOX DRAWINGS DOWN DOUBLE AND LEFT SINGLE
184	0x2555	‘’	BOX DRAWINGS DOWN SINGLE AND LEFT DOUBLE
185	0x2563	‘’	BOX DRAWINGS DOUBLE VERTICAL AND LEFT
186	0x2551	‘’	BOX DRAWINGS DOUBLE VERTICAL
187	0x2557	‘’	BOX DRAWINGS DOUBLE DOWN AND LEFT
188	0x255D	‘’	BOX DRAWINGS DOUBLE UP AND LEFT
189	0x255C	‘’	BOX DRAWINGS UP DOUBLE AND LEFT SINGLE
190	0x255B	‘’	BOX DRAWINGS UP SINGLE AND LEFT DOUBLE
191	0x2510	‘’	BOX DRAWINGS LIGHT DOWN AND LEFT
192	0x2514	‘ ’	BOX DRAWINGS LIGHT UP AND RIGHT
193	0x2534	‘’	BOX DRAWINGS LIGHT UP AND HORIZONTAL
194	0x252C	‘’	BOX DRAWINGS LIGHT DOWN AND HORIZONTAL
195	0x251C	‘ ’	BOX DRAWINGS LIGHT VERTICAL AND RIGHT
196	0x2500	‘ ’	BOX DRAWINGS LIGHT HORIZONTAL
197	0x253C	‘’	BOX DRAWINGS LIGHT VERTICAL AND HORIZONTAL

Continued on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
198	0x255E	‘’	BOX DRAWINGS VERTICAL SINGLE AND RIGHT DOUBLE
199	0x255F	‘’	BOX DRAWINGS VERTICAL DOUBLE AND RIGHT SINGLE
200	0x255A	‘’	BOX DRAWINGS DOUBLE UP AND RIGHT
201	0x2554	‘’	BOX DRAWINGS DOUBLE DOWN AND RIGHT
202	0x2569	‘’	BOX DRAWINGS DOUBLE UP AND HORIZONTAL
203	0x2566	‘’	BOX DRAWINGS DOUBLE DOWN AND HORIZONTAL
204	0x2560	‘’	BOX DRAWINGS DOUBLE VERTICAL AND RIGHT
205	0x2550	‘’	BOX DRAWINGS DOUBLE HORIZONTAL
206	0x256C	‘’	BOX DRAWINGS DOUBLE VERTICAL AND HORIZONTAL
207	0x2567	‘’	BOX DRAWINGS UP SINGLE AND HORIZONTAL DOUBLE
208	0x2568	‘’	BOX DRAWINGS UP DOUBLE AND HORIZONTAL SINGLE
209	0x2564	‘’	BOX DRAWINGS DOWN SINGLE AND HORIZONTAL DOUBLE
210	0x2565	‘’	BOX DRAWINGS DOWN DOUBLE AND HORIZONTAL SINGLE
211	0x2559	‘’	BOX DRAWINGS UP DOUBLE AND RIGHT SINGLE
212	0x2558	‘’	BOX DRAWINGS UP SINGLE AND RIGHT DOUBLE
213	0x2552	‘’	BOX DRAWINGS DOWN SINGLE AND RIGHT DOUBLE
214	0x2553	‘’	BOX DRAWINGS DOWN DOUBLE AND RIGHT SINGLE
215	0x256B	‘’	BOX DRAWINGS VERTICAL DOUBLE AND HORIZONTAL SINGLE
216	0x256A	‘’	BOX DRAWINGS VERTICAL SINGLE AND HORIZONTAL DOUBLE
217	0x2518	‘’	BOX DRAWINGS LIGHT UP AND LEFT
218	0x250C	‘’	BOX DRAWINGS LIGHT DOWN AND RIGHT
219	0x2588	‘’	FULL BLOCK
220	0x2584	‘’	LOWER HALF BLOCK
221	0x258C	‘’	LEFT HALF BLOCK
222	0x2590	‘’	RIGHT HALF BLOCK
223	0x2580	‘’	UPPER HALF BLOCK
224	0x03B1	‘α’	GREEK SMALL LETTER ALPHA
225	0xDF	‘ß’	LATIN SMALL LETTER SHARP S
226	0x0393	‘Γ’	GREEK CAPITAL LETTER GAMMA
227	0x03C0	‘π’	GREEK SMALL LETTER PI
228	0x03A3	‘Σ’	GREEK CAPITAL LETTER SIGMA
229	0x03C3	‘σ’	GREEK SMALL LETTER SIGMA
230	0xB5	‘µ’	MICRO SIGN
231	0x03C4	‘τ’	GREEK SMALL LETTER TAU
232	0x03A6	‘Φ’	GREEK CAPITAL LETTER PHI
233	0x0398	‘Θ’	GREEK CAPITAL LETTER THETA
234	0x03A9	‘Ω’	GREEK CAPITAL LETTER OMEGA
235	0x03B4	‘δ’	GREEK SMALL LETTER DELTA
236	0x221E	‘∞’	INFINITY
237	0x03C6	‘ϕ’	GREEK SMALL LETTER PHI
238	0x03B5	‘ϵ’	GREEK SMALL LETTER EPSILON
239	0x2229	‘’	INTERSECTION
240	0x2261	‘’	IDENTICAL TO
241	0xB1	‘±’	PLUS-MINUS SIGN
242	0x2265	‘’	GREATER-THAN OR EQUAL TO
243	0x2264	‘’	LESS-THAN OR EQUAL TO
244	0x2320	‘’	TOP HALF INTEGRAL
245	0x2321	‘’	BOTTOM HALF INTEGRAL
246	0xF7	‘÷’	DIVISION SIGN

Continued on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
247	0x2248	‘’	ALMOST EQUAL TO
248	0xB0	‘°’	DEGREE SIGN
249	0x2219	‘’	BULLET OPERATOR
250	0xB7	‘·’	MIDDLE DOT
251	0x221A	‘’	SQUARE ROOT
252	0x207F	‘’	SUPERSCRIPT LATIN SMALL LETTER N
253	0xB2	‘²’	SUPERSCRIPT TWO
254	0x25A0	‘’	BLACK SQUARE
255	0xA0	‘\xa0’	NO-BREAK SPACE

5.2 Deprecated TCOD Layout

The layout for tilesets loaded with: `tcod.tileset.CHARMAP_TCOD`

Tile Index	Unicode	String	Name
0	0x20	‘ ‘	SPACE
1	0x21	‘!’	EXCLAMATION MARK
2	0x22	‘”’	QUOTATION MARK
3	0x23	‘#’	NUMBER SIGN
4	0x24	‘$’	DOLLAR SIGN
5	0x25	‘%’	PERCENT SIGN
6	0x26	‘&’	AMPERSAND
7	0x27	“’”	APOSTROPHE
8	0x28	‘(‘	LEFT PARENTHESIS
9	0x29	‘)’	RIGHT PARENTHESIS
10	0x2A	‘*’	ASTERISK
11	0x2B	‘+’	PLUS SIGN
12	0x2C	‘,’	COMMA
13	0x2D	‘-‘	HYPHEN-MINUS
14	0x2E	‘.’	FULL STOP
15	0x2F	‘/’	SOLIDUS
16	0x30	‘0’	DIGIT ZERO
17	0x31	‘1’	DIGIT ONE
18	0x32	‘2’	DIGIT TWO
19	0x33	‘3’	DIGIT THREE
20	0x34	‘4’	DIGIT FOUR
21	0x35	‘5’	DIGIT FIVE
22	0x36	‘6’	DIGIT SIX
23	0x37	‘7’	DIGIT SEVEN
24	0x38	‘8’	DIGIT EIGHT
25	0x39	‘9’	DIGIT NINE
26	0x3A	‘:’	COLON
27	0x3B	‘;’	SEMICOLON
28	0x3C	‘<’	LESS-THAN SIGN
29	0x3D	‘=’	EQUALS SIGN
30	0x3E	‘>’	GREATER-THAN SIGN
31	0x3F	‘?’	QUESTION MARK

Continued on next page

Table 2 – continued from previous page

Tile Index	Unicode	String	Name
32	0x40	'@'	COMMERCIAL AT
33	0x5B	'['	LEFT SQUARE BRACKET
34	0x5C	''	REVERSE SOLIDUS
35	0x5D	']'	RIGHT SQUARE BRACKET
36	0x5E	'^'	CIRCUMFLEX ACCENT
37	0x5F	'_'	LOW LINE
38	0x60	''	GRAVE ACCENT
39	0x7B	'{'	LEFT CURLY BRACKET
40	0x7C	'|'	VERTICAL LINE
41	0x7D	'}'	RIGHT CURLY BRACKET
42	0x7E	'~'	TILDE
43	0x2591	''	LIGHT SHADE
44	0x2592	''	MEDIUM SHADE
45	0x2593	''	DARK SHADE
46	0x2502	''	BOX DRAWINGS LIGHT VERTICAL
47	0x2500	''	BOX DRAWINGS LIGHT HORIZONTAL
48	0x253C	''	BOX DRAWINGS LIGHT VERTICAL AND HORIZONTAL
49	0x2524	''	BOX DRAWINGS LIGHT VERTICAL AND LEFT
50	0x2534	''	BOX DRAWINGS LIGHT UP AND HORIZONTAL
51	0x251C	''	BOX DRAWINGS LIGHT VERTICAL AND RIGHT
52	0x252C	''	BOX DRAWINGS LIGHT DOWN AND HORIZONTAL
53	0x2514	''	BOX DRAWINGS LIGHT UP AND RIGHT
54	0x250C	''	BOX DRAWINGS LIGHT DOWN AND RIGHT
55	0x2510	''	BOX DRAWINGS LIGHT DOWN AND LEFT
56	0x2518	''	BOX DRAWINGS LIGHT UP AND LEFT
57	0x2598	''	QUADRANT UPPER LEFT
58	0x259D	''	QUADRANT UPPER RIGHT
59	0x2580	''	UPPER HALF BLOCK
60	0x2596	''	QUADRANT LOWER LEFT
61	0x259A	''	QUADRANT UPPER LEFT AND LOWER RIGHT
62	0x2590	''	RIGHT HALF BLOCK
63	0x2597	''	QUADRANT LOWER RIGHT
64	0x2191	'↑'	UPWARDS ARROW
65	0x2193	'↓'	DOWNWARDS ARROW
66	0x2190	'←'	LEFTWARDS ARROW
67	0x2192	'→'	RIGHTWARDS ARROW
68	0x25B2	''	BLACK UP-POINTING TRIANGLE
69	0x25BC	''	BLACK DOWN-POINTING TRIANGLE
70	0x25C4	''	BLACK LEFT-POINTING POINTER
71	0x25BA	''	BLACK RIGHT-POINTING POINTER
72	0x2195	''	UP DOWN ARROW
73	0x2194	''	LEFT RIGHT ARROW
74	0x2610	''	BALLOT BOX
75	0x2611	''	BALLOT BOX WITH CHECK
76	0x25CB	''	WHITE CIRCLE
77	0x25C9	''	FISHEYE
78	0x2551	''	BOX DRAWINGS DOUBLE VERTICAL
79	0x2550	''	BOX DRAWINGS DOUBLE HORIZONTAL
80	0x256C	''	BOX DRAWINGS DOUBLE VERTICAL AND HORIZONTAL

Continued on next page

Table 2 – continued from previous page

Tile Index	Unicode	String	Name
81	0x2563	‘’	BOX DRAWINGS DOUBLE VERTICAL AND LEFT
82	0x2569	‘’	BOX DRAWINGS DOUBLE UP AND HORIZONTAL
83	0x2560	‘’	BOX DRAWINGS DOUBLE VERTICAL AND RIGHT
84	0x2566	‘’	BOX DRAWINGS DOUBLE DOWN AND HORIZONTAL
85	0x255A	‘’	BOX DRAWINGS DOUBLE UP AND RIGHT
86	0x2554	‘’	BOX DRAWINGS DOUBLE DOWN AND RIGHT
87	0x2557	‘’	BOX DRAWINGS DOUBLE DOWN AND LEFT
88	0x255D	‘’	BOX DRAWINGS DOUBLE UP AND LEFT
89	0x00	‘\x00’	
90	0x00	‘\x00’	
91	0x00	‘\x00’	
92	0x00	‘\x00’	
93	0x00	‘\x00’	
94	0x00	‘\x00’	
95	0x00	‘\x00’	
96	0x41	‘A’	LATIN CAPITAL LETTER A
97	0x42	‘B’	LATIN CAPITAL LETTER B
98	0x43	‘C’	LATIN CAPITAL LETTER C
99	0x44	‘D’	LATIN CAPITAL LETTER D
100	0x45	‘E’	LATIN CAPITAL LETTER E
101	0x46	‘F’	LATIN CAPITAL LETTER F
102	0x47	‘G’	LATIN CAPITAL LETTER G
103	0x48	‘H’	LATIN CAPITAL LETTER H
104	0x49	‘I’	LATIN CAPITAL LETTER I
105	0x4A	‘J’	LATIN CAPITAL LETTER J
106	0x4B	‘K’	LATIN CAPITAL LETTER K
107	0x4C	‘L’	LATIN CAPITAL LETTER L
108	0x4D	‘M’	LATIN CAPITAL LETTER M
109	0x4E	‘N’	LATIN CAPITAL LETTER N
110	0x4F	‘O’	LATIN CAPITAL LETTER O
111	0x50	‘P’	LATIN CAPITAL LETTER P
112	0x51	‘Q’	LATIN CAPITAL LETTER Q
113	0x52	‘R’	LATIN CAPITAL LETTER R
114	0x53	‘S’	LATIN CAPITAL LETTER S
115	0x54	‘T’	LATIN CAPITAL LETTER T
116	0x55	‘U’	LATIN CAPITAL LETTER U
117	0x56	‘V’	LATIN CAPITAL LETTER V
118	0x57	‘W’	LATIN CAPITAL LETTER W
119	0x58	‘X’	LATIN CAPITAL LETTER X
120	0x59	‘Y’	LATIN CAPITAL LETTER Y
121	0x5A	‘Z’	LATIN CAPITAL LETTER Z
122	0x00	‘\x00’	
123	0x00	‘\x00’	
124	0x00	‘\x00’	
125	0x00	‘\x00’	
126	0x00	‘\x00’	
127	0x00	‘\x00’	
128	0x61	‘a’	LATIN SMALL LETTER A
129	0x62	‘b’	LATIN SMALL LETTER B

Continued on next page

Table 2 – continued from previous page

Tile Index	Unicode	String	Name
130	0x63	‘c’	LATIN SMALL LETTER C
131	0x64	‘d’	LATIN SMALL LETTER D
132	0x65	‘e’	LATIN SMALL LETTER E
133	0x66	‘f’	LATIN SMALL LETTER F
134	0x67	‘g’	LATIN SMALL LETTER G
135	0x68	‘h’	LATIN SMALL LETTER H
136	0x69	‘i’	LATIN SMALL LETTER I
137	0x6A	‘j’	LATIN SMALL LETTER J
138	0x6B	‘k’	LATIN SMALL LETTER K
139	0x6C	‘l’	LATIN SMALL LETTER L
140	0x6D	‘m’	LATIN SMALL LETTER M
141	0x6E	‘n’	LATIN SMALL LETTER N
142	0x6F	‘o’	LATIN SMALL LETTER O
143	0x70	‘p’	LATIN SMALL LETTER P
144	0x71	‘q’	LATIN SMALL LETTER Q
145	0x72	‘r’	LATIN SMALL LETTER R
146	0x73	‘s’	LATIN SMALL LETTER S
147	0x74	‘t’	LATIN SMALL LETTER T
148	0x75	‘u’	LATIN SMALL LETTER U
149	0x76	‘v’	LATIN SMALL LETTER V
150	0x77	‘w’	LATIN SMALL LETTER W
151	0x78	‘x’	LATIN SMALL LETTER X
152	0x79	‘y’	LATIN SMALL LETTER Y
153	0x7A	‘z’	LATIN SMALL LETTER Z
154	0x00	‘\x00’	
155	0x00	‘\x00’	
156	0x00	‘\x00’	
157	0x00	‘\x00’	
158	0x00	‘\x00’	
159	0x00	‘\x00’	

CHAPTER 6

tcod.bsp - Binary Space Partitioning

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

Example:

```
import tcod

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,
)

# In pre order, leaf nodes are visited before the nodes that connect them.
for node in bsp.pre_order():
    if node.children:
        node1, node2 = node.children
        print('Connect the rooms:\n%s\n%s' % (node1, node2))
    else:
        print('Dig a room for %s.' % node)
```

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

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

x

Rectangle left coordinate.

Type int

y

Rectangle top coordinate.

Type int

width
Rectangle width.

Type int

height
Rectangle height.

Type int

level
This nodes depth.

Type int

position
The integer of where the node was split.

Type int

horizontal
This nodes split orientation.

Type bool

parent
This nodes parent or None

Type Optional[*BSP*]

children
A tuple of (left, right) BSP instances, or an empty tuple if this BSP has no children.

Type Union[Tuple[()], Tuple[*BSP*, *BSP*]]

Parameters

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

__str__() → str
Provide a useful readout when printed.

contains(*x: int*, *y: int*) → bool
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: int*, *y: int*) → Optional[tcod.bsp.BSP]
Return the deepest node which contains these coordinates.

Returns BSP object or None.

Return type Optional[*BSP*]

in_order() → Iterator[tcod.bsp.BSP]
Iterate over this BSP's hierarchy in order.

New in version 8.3.

inverted_level_order() → Iterator[tcod.bsp.BSP]
Iterate over this BSP's hierarchy in inverse level order.

New in version 8.3.

level_order() → Iterator[tcod.bsp.BSP]
Iterate over this BSP's hierarchy in level order.

New in version 8.3.

post_order() → Iterator[tcod.bsp.BSP]
Iterate over this BSP's hierarchy in post order.

New in version 8.3.

pre_order() → Iterator[tcod.bsp.BSP]
Iterate over this BSP's hierarchy in pre order.

New in version 8.3.

split_once(*horizontal: bool*, *position: int*) → None
Split this partition into 2 sub-partitions.

Parameters

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

split_recursive(*depth: int*, *min_width: int*, *min_height: int*, *max_horizontal_ratio: float*, *max_vertical_ratio: float*, *seed: Optional[tcod.random.Random] = None*) → 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() → Iterator[tcod.bsp.BSP]
Iterate over this BSP's hierarchy in pre order.

Deprecated since version 2.3: Use *pre_order* instead.

CHAPTER 7

tcod.console - Tile Drawing/Printing

Libtcod consoles are a strictly tile-based representation of text and color. To render a console you need a tileset and a window to render to. See *Getting Started* for info on how to set those up.

class `tcod.console.`**`Console`**(*width: int, height: int, order: Union[typing_extensions.Literal['C'][C], typing_extensions.Literal['F'][F]] = 'C', buffer: Optional[numpy.ndarray] = None*)

A console object containing a grid of characters with foreground/background colors.

width and *height* are the size of the console (in tiles.)

order determines how the axes of NumPy array attributes are arraigned. *order="F"* will swap the first two axes which allows for more intuitive *[x, y]* indexing.

With *buffer* the console can be initialized from another array. The *buffer* should be compatible with the *width*, *height*, and *order* given; and should also have a dtype compatible with `Console.DTYPE`.

Changed in version 4.3: Added *order* parameter.

Changed in version 8.5: Added *buffer*, *copy*, and default parameters. Arrays are initialized as if the `clear` method was called.

Changed in version 10.0: *DTYPE* changed, *buffer* now requires colors with an alpha channel.

console_c

A python-cffi "TCOD_Console*" object.

DTYPE

A class attribute which provides a dtype compatible with this class.

`[("ch", np.intc), ("fg", "(4,)u1"), ("bg", "(4,)u1")]`

Example:

```
>>> buffer = np.zeros(
...     shape=(20, 3),
...     dtype=tcod.console.Console.DTYPE,
...     order="F",
```

(continues on next page)

(continued from previous page)

```
... )
>>> buffer["ch"] = ord(' ')
>>> buffer["ch"][:, 1] = ord('x')
>>> c = tcod.console.Console(20, 3, order="F", buffer=buffer)
>>> print(c)
<                    |
|xxxxxxxxxxxxxxxxxxx|
|                    >
```

New in version 8.5.

Changed in version 10.0: Added an alpha channel to the color types.

__bool__() → bool

Returns False if this is the root console.

This mimics libtcodpy behavior.

__enter__() → tcod.console.Console

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.

See also:

`tcod.console_init_root`

__exit__(**args*) → None

Closes the graphical window on exit.

Some tcod functions may have undefined behavior after this point.

__repr__() → str

Return a string representation of this console.

__str__() → str

Return a simplified representation of this consoles contents.

blit(*dest: tcod.console.Console, dest_x: int = 0, dest_y: int = 0, src_x: int = 0, src_y: int = 0, width: int = 0, height: int = 0, fg_alpha: float = 1.0, bg_alpha: float = 1.0, key_color: Optional[Tuple[int, int, int]] = None*) → None

Blit from this console onto the `dest` console.

Parameters

- **dest** (`Console`) – The destination console to blit onto.
- **dest_x** (`int`) – Leftmost coordinate of the destination console.
- **dest_y** (`int`) – Topmost coordinate of the destination 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, *)*

Changed in version 11.6: Now supports per-cell alpha transparency.

Use `Console.buffer` to set tile alpha before blit.

clear(*ch: int = 32*, *fg: Tuple[int, int, int] = Ellipsis*, *bg: Tuple[int, int, int] = Ellipsis*) → None

Reset all values in this console to a single value.

ch is the character to clear the console with. Defaults to the space character.

fg and *bg* are the colors to clear the console with. Defaults to white-on-black if the console defaults are untouched.

Note: If *fg*/*bg* are not set, they will default to `default_fg`/`default_bg`. However, default values other than white-on-back are deprecated.

Changed in version 8.5: Added the *ch*, *fg*, and *bg* parameters. Non-white-on-black default values are deprecated.

close() → None

Close the active window managed by libtcod.

This must only be called on the root console, which is returned from `tcod.console_init_root`.

New in version 11.11.

draw_frame(*x: int*, *y: int*, *width: int*, *height: int*, *title: str = ''*, *clear: bool = True*, *fg: Optional[Tuple[int, int, int]] = None*, *bg: Optional[Tuple[int, int, int]] = None*, *bg_blend: int = 1*) → None

Draw a framed rectangle with an optional title.

x and *y* are the starting tile, with `0,0` as the upper-left corner of the console. You can use negative numbers if you want to start printing relative to the bottom-right corner, but this behavior may change in future versions.

width and *height* determine the size of the frame.

title is a Unicode string. The title is drawn with *bg* as the text color and *fg* as the background.

If *clear* is True than the region inside of the frame will be cleared.

fg and *bg* are the foreground and background colors for the frame border. This is a 3-item tuple with (r, g, b) color values from 0 to 255. These parameters can also be set to *None* to leave the colors unchanged.

bg_blend is the blend type used by libtcod.

New in version 8.5.

Changed in version 9.0: *fg* and *bg* now default to *None* instead of white-on-black.

draw_rect(*x: int*, *y: int*, *width: int*, *height: int*, *ch: int*, *fg: Optional[Tuple[int, int, int]] = None*, *bg: Optional[Tuple[int, int, int]] = None*, *bg_blend: int = 1*) → None

Draw characters and colors over a rectangular region.

x and *y* are the starting tile, with `0,0` as the upper-left corner of the console. You can use negative numbers if you want to start printing relative to the bottom-right corner, but this behavior may change in future versions.

width and *height* determine the size of the rectangle.

ch is a Unicode integer. You can use 0 to leave the current characters unchanged.

fg and *bg* are the foreground text color and background tile color respectfully. This is a 3-item tuple with (r, g, b) color values from 0 to 255. These parameters can also be set to *None* to leave the colors unchanged.

bg_blend is the blend type used by libtcod.

New in version 8.5.

Changed in version 9.0: *fg* and *bg* now default to *None* instead of white-on-black.

draw_semigraphics(*pixels: Any*, *x: int = 0*, *y: int = 0*) → None

Draw a block of 2x2 semi-graphics into this console.

pixels is an Image or an array-like object. It will be down-sampled into 2x2 blocks when drawn. Array-like objects must be in the shape of *(height, width, RGB)* and should have a *dtype* of *numpy.uint8*.

x and *y* is the upper-left tile position to start drawing.

New in version 11.4.

get_height_rect(*x: int*, *y: int*, *width: int*, *height: int*, *string: str*) → int

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** (*str*) – A Unicode string.

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

Return type int

hline(*x: int*, *y: int*, *width: int*, *bg_blend: int = 13*) → None

Draw a horizontal line on the console.

This always uses ord('–'), the horizontal line character.

Parameters

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

Deprecated since version 8.5: Console methods which depend on console defaults have been deprecated. Use `Console.draw_rect` instead, calling this function will print a warning detailing which default values need to be made explicit.

print(*x: int*, *y: int*, *string: str*, *fg: Optional[Tuple[int, int, int]] = None*, *bg: Optional[Tuple[int, int, int]] = None*, *bg_blend: int = 1*, *alignment: int = 0*) → None

Print a string on a console with manual line breaks.

x and *y* are the starting tile, with `0,0` as the upper-left corner of the console. You can use negative numbers if you want to start printing relative to the bottom-right corner, but this behavior may change in future versions.

string is a Unicode string which may include color control characters. Strings which are too long will be truncated until the next newline character `"\n"`.

fg and *bg* are the foreground text color and background tile color respectfully. This is a 3-item tuple with (r, g, b) color values from 0 to 255. These parameters can also be set to *None* to leave the colors unchanged.

bg_blend is the blend type used by libtcod.

alignment can be *tcod.LEFT*, *tcod.CENTER*, or *tcod.RIGHT*.

New in version 8.5.

Changed in version 9.0: *fg* and *bg* now default to *None* instead of white-on-black.

print_(*x: int*, *y: int*, *string: str*, *bg_blend: int = 13*, *alignment: Optional[int] = None*) → 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** (*str*) – A Unicode string optionally using color codes.
- **bg_blend** (*int*) – Blending mode to use, defaults to BKGND_DEFAULT.
- **alignment** (*Optional[int]*) – Text alignment.

Deprecated since version 8.5: Console methods which depend on console defaults have been deprecated. Use *Console.print* instead, calling this function will print a warning detailing which default values need to be made explicit.

print_box(*x: int*, *y: int*, *width: int*, *height: int*, *string: str*, *fg: Optional[Tuple[int, int, int]] = None*, *bg: Optional[Tuple[int, int, int]] = None*, *bg_blend: int = 1*, *alignment: int = 0*) → int

Print a string constrained to a rectangle and return the height.

x and *y* are the starting tile, with `0,0` as the upper-left corner of the console. You can use negative numbers if you want to start printing relative to the bottom-right corner, but this behavior may change in future versions.

width and *height* determine the bounds of the rectangle, the text will automatically be word-wrapped to fit within these bounds.

string is a Unicode string which may include color control characters.

fg and *bg* are the foreground text color and background tile color respectfully. This is a 3-item tuple with (r, g, b) color values from 0 to 255. These parameters can also be set to *None* to leave the colors unchanged.

bg_blend is the blend type used by libtcod.

alignment can be *tcod.LEFT*, *tcod.CENTER*, or *tcod.RIGHT*.

Returns the actual height of the printed area.

New in version 8.5.

Changed in version 9.0: *fg* and *bg* now default to *None* instead of white-on-black.

print_frame(*x: int*, *y: int*, *width: int*, *height: int*, *string: str = ''*, *clear: bool = True*, *bg_blend: int = 13*) → None

Draw a framed rectangle with optional 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. If *string* is empty then no title will be drawn.

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** (*str*) – 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.

Changed in version 8.2: Now supports Unicode strings.

Deprecated since version 8.5: Console methods which depend on console defaults have been deprecated. Use `Console.draw_frame` instead, calling this function will print a warning detailing which default values need to be made explicit.

print_rect(*x: int*, *y: int*, *width: int*, *height: int*, *string: str*, *bg_blend: int = 13*, *alignment: Optional[int] = None*) → int

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** (*str*) – 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

Deprecated since version 8.5: Console methods which depend on console defaults have been deprecated. Use `Console.print_box` instead, calling this function will print a warning detailing which default values need to be made explicit.

put_char(*x: int*, *y: int*, *ch: int*, *bg_blend: int = 13*) → None

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: int*, *y: int*, *width: int*, *height: int*, *clear: bool*, *bg_blend: int = 13*) → None

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

If *clear* 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.

Deprecated since version 8.5: Console methods which depend on console defaults have been deprecated. Use `Console.draw_rect` instead, calling this function will print a warning detailing which default values need to be made explicit.

set_key_color(*color: Optional[Tuple[int, int, int]]*) → None

Set a consoles blit transparent color.

color is the (r, g, b) color, or None to disable key color.

Deprecated since version 8.5: Pass the key color to `Console.blit` instead of calling this function.

vline(*x: int*, *y: int*, *height: int*, *bg_blend: int = 13*) → None

Draw a vertical line on the console.

This always uses ord('│'), 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.

Deprecated since version 8.5: Console methods which depend on console defaults have been deprecated. Use `Console.draw_rect` instead, calling this function will print a warning detailing which default values need to be made explicit.

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[i, j, channel] # order='C'` or `console.bg[x, y, channel] # order='F'`.

buffer

An array of this consoles raw tile data.

New in version 11.4.

Deprecated since version 11.8: Use `Console.tiles` instead.

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[i, j] # order='C'` or `console.ch[x, y] # order='F'`.

default_alignment

The default text alignment.

Type int

default_bg

The default background color.

Type Tuple[int, int, int]

default_bg_blend

The default blending mode.

Type int

default_fg

The default foreground color.

Type Tuple[int, int, int]

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[i, j, channel] # order='C'` or `console.fg[x, y, channel] # order='F'`.

height

The height of this Console. (read-only)

Type int

tiles

An array of this consoles raw tile data.

This acts as a combination of the *ch*, *fg*, and *bg* attributes. Colors include an alpha channel but how alpha works is currently undefined.

Example

```
>>> con = tcod.console.Console(10, 2)
>>> con.tiles[0, 0] = (
...     ord("X"),
...     (*tcod.white, 255),
...     (*tcod.black, 255),
... )
>>> con.tiles[0, 0]
(88, [255, 255, 255, 255], [  0,   0,   0, 255])
```

New in version 10.0.

tiles2

This name is deprecated in favour of *tiles_rgb*.

New in version 11.3.

Deprecated since version 11.8: Use *Console.tiles_rgb* instead.

tiles_rgb

An array of this consoles tile data without the alpha channel.

The dtype of this array is effectively: `[("ch", np.intc), ("fg", "(3,)u1"), ("bg", "(3,)u1")]`

Example

```
>>> con = tcod.console.Console(10, 2)
>>> con.tiles_rgb[0, 0] = ord("@"), tcod.yellow, tcod.black
>>> con.tiles_rgb[0, 0]
(64, [255, 255,   0], [0, 0, 0])
>>> con.tiles_rgb["bg"] = tcod.blue
>>> con.tiles_rgb[0, 0]
(64, [255, 255,   0], [  0,   0, 255])
```

New in version 11.8.

width

The width of this Console. (read-only)

Type int

`tcod.console.`**`get_height_rect`**(*width: int*, *string: str*) → int

Return the number of lines which would be printed from these parameters.

width is the width of the print boundary.

string is a Unicode string which may include color control characters.

New in version 9.2.

`tcod.console.`**`recommended_size`**() → Tuple[int, int]

Return the recommended size of a console for the current active window.

The return is determined from the active tileset size and active window size. This result should be used create an *Console* instance.

This function will raise RuntimeError if libtcod has not been initialized.

New in version 11.8.

See also:

tcod.console_init_root *tcod.console_flush*

Deprecated since version 11.13: This function does not support contexts. Use *Context.recommended_console_size* instead.

CHAPTER 8

tcod.context - Window Management

This module is used to create and handle libtcod contexts.

See *Getting Started* for beginner examples on how to use this module.

`Context`'s are intended to replace several libtcod functions such as `tcod.console_init_root`, `tcod.console_flush`, `tcod.console.recommended_size`, and many other functions which rely on hidden global objects within libtcod. If you begin using contexts then most of these functions will no longer work properly.

Instead of calling `tcod.console_init_root` you can call `tcod.context.new` with different keywords depending on how you plan to setup the size of the console. You should use `tcod.tileset` to load the font for a context.

Note: If you use contexts then you should expect the following functions to no longer be available, because these functions rely on a global console, tileset, or some other kind of global state:

`tcod.console_init_root`, `tcod.console_set_custom_font`, `tcod.console_flush`, `tcod.console_is_fullscreen`, `tcod.console_is_window_closed`, `tcod.console_set_fade`, `tcod.console_set_fullscreen`, `tcod.console_set_window_title`, `tcod.sys_set_fps`, `tcod.sys_get_last_frame_length`, `tcod.sys_set_renderer`, `tcod.sys_save_screenshot`, `tcod.sys_force_fullscreen_resolution`, `tcod.sys_get_current_resolution`, `tcod.sys_register_SDL_renderer`, `tcod.console_map_ascii_code_to_font`, `tcod.sys_get_char_size`, `tcod.sys_update_char`, `tcod.console.recommended_size`, `tcod.tileset.get_default`, `tcod.tileset.set_default`.

Some event functions can no longer return tile coordinates for the mouse: `tcod.sys_check_for_event`, `tcod.sys_wait_for_event`, `tcod.mouse_get_status`.

New in version 11.12.

class `tcod.context.`**`Context`**(*context_p: Any*)

Context manager for libtcod context objects.

You should use `tcod.context.new_terminal` or `tcod.context.new_window` to create a new context.

__enter__() → tcod.context.Context
This context can be used as a context manager.

__exit__(**args*) → None
The libtcod context is closed as this context manager exits.

change_tileset(*tileset: Optional[tcod.tileset.Tileset]*) → None
Change the active tileset used by this context.

close() → None
Delete the context, closing any windows opened by this context.

This instance is invalid after this call.

convert_event(*event: tcod.event.Event*) → None
Fill in the tile coordinates of a mouse event using this context.

new_console(*min_columns: int = 1, min_rows: int = 1, magnification: float = 1.0, order: Union[typing_extensions.Literal['C'][C], typing_extensions.Literal['F'][F]] = 'C'*) → tcod.console.Console
Return a new console sized for this context.

min_columns and *min_rows* are the minimum size to use for the new console.

magnification determines the apparent size of the tiles on the output display. A *magnification* larger then 1.0 will output smaller consoles, which will show as larger tiles when presented. *magnification* must be greater than zero.

order is passed to `tcod.console.Console` to determine the memory order of its NumPy attributes.

The times where it is the most useful to call this method are:

- After the context is created, even if the console was given a specific size.
- After the `change_tileset` method is called.
- After any window resized event, or any manual resizing of the window.

New in version 11.18.

Changed in version 11.19: Added *order* parameter.

See also:

`tcod.console.Console`

pixel_to_subtile(*x: int, y: int*) → Tuple[float, float]
Convert window pixel coordinates to sub-tile coordinates.

pixel_to_tile(*x: int, y: int*) → Tuple[int, int]
Convert window pixel coordinates to tile coordinates.

present(*console: tcod.console.Console, *, keep_aspect: bool = False, integer_scaling: bool = False, clear_color: Tuple[int, int, int] = (0, 0, 0), align: Tuple[float, float] = (0.5, 0.5)*) → None
Present a console to this context's display.

console is the console you want to present.

If *keep_aspect* is True then the console aspect will be preserved with a letterbox. Otherwise the console will be stretched to fill the screen.

If *integer_scaling* is True then the console will be scaled in integer increments. This will have no effect if the console must be shrunk. You can use `tcod.console.recommended_size` to create a console which will fit the window without needing to be scaled.

clear_color is an RGB tuple used to clear the screen before the console is presented, this will affect the border/letterbox color.

align is an (x, y) tuple determining where the console will be placed when letter-boxing exists. Values of 0 will put the console at the upper-left corner. Values of 0.5 will center the console.

recommended_console_size(*min_columns: int = 1*, *min_rows: int = 1*) → Tuple[int, int]

Return the recommended (columns, rows) of a console for this context.

min_columns, *min_rows* are the lowest values which will be returned.

If result is only used to create a new console then you may want to call `Context.new_console` instead.

save_screenshot(*path: Optional[str] = None*) → None

Save a screen-shot to the given file path.

renderer_type

Return the libtcod renderer type used by this context.

sdl_window_p

A cffi *SDL_Window** pointer. This pointer might be NULL.

This pointer will become invalid if the context is closed or goes out of scope.

Python-tcod's FFI provides most SDL functions. So it's possible for anyone with the SDL2 documentation to work directly with SDL's pointers.

Example:

```
import tcod

def toggle_fullscreen(context: tcod.context.Context) -> None:
    """Toggle a context window between fullscreen and windowed modes."""
    if not context.sdl_window_p:
        return
    fullscreen = tcod.lib.SDL_GetWindowFlags(context.sdl_window_p) & (
        tcod.lib.SDL_WINDOW_FULLSCREEN | tcod.lib.SDL_WINDOW_FULLSCREEN_
↪DESKTOP
    )
    tcod.lib.SDL_SetWindowFullscreen(
        context.sdl_window_p,
        0 if fullscreen else tcod.lib.SDL_WINDOW_FULLSCREEN_DESKTOP,
    )
```

tcod.context.**new**(*, *x: Optional[int] = None*, *y: Optional[int] = None*, *width: Optional[int] = None*, *height: Optional[int] = None*, *columns: Optional[int] = None*, *rows: Optional[int] = None*, *renderer: Optional[int] = None*, *tileset: Optional[tcod.tileset.Tileset] = None*, *vsync: bool = True*, *sdl_window_flags: Optional[int] = None*, *title: Optional[str] = None*, *argv: Optional[Iterable[str]] = None*) → tcod.context.Context

Create a new context with the desired pixel size.

x, *y*, *width*, and *height* are the desired position and size of the window. If these are None then they will be derived from *columns* and *rows*. So if you plan on having a console of a fixed size then you should set *columns* and *rows* instead of the window keywords.

columns and *rows* is the desired size of the console. Can be left as *None* when you're setting a context by a window size instead of a console.

Providing no size information at all is also acceptable.

renderer is the desired libtcod renderer to use. Typical options are `tcod.context.RENDERER_OPENGL2` for a faster renderer or `tcod.context.RENDERER_SDL2` for a reliable renderer.

tileset is the font/tileset for the new context to render with. The fall-back tileset available from passing None is useful for prototyping, but will be unreliable across platforms.

vsync is the Vertical Sync option for the window. The default of True is recommended but you may want to use False for benchmarking purposes.

sdl_window_flags is a bit-field of SDL window flags, if None is given then a default of `tcod.context.SDL_WINDOW_RESIZABLE` is used. There's more info on the SDL documentation: https://wiki.libsdl.org/SDL_CreateWindow#Remarks

title is the desired title of the window.

argv these arguments are passed to libtcod and allow an end-user to make last second changes such as forcing fullscreen or windowed mode, or changing the libtcod renderer. By default this will pass in *sys.argv* but you can disable this feature by providing an empty list instead. Certain commands such as `-help` will raise a SystemExit exception from this function with the output message.

When a window size is given instead of a console size then you can use `Context.recommended_console_size` to automatically find the size of the console which should be used.

New in version 11.16.

`tcod.context.`**`new_window`**(*width: int*, *height: int*, ***, *renderer: Optional[int] = None*, *tileset: Optional[tcod.tileset.Tileset] = None*, *vsync: bool = True*, *sdl_window_flags: Optional[int] = None*, *title: Optional[str] = None*) → tcod.context.Context

Create a new context with the desired pixel size.

Deprecated since version 11.16: `tcod.context.new` provides more options, such as window position.

`tcod.context.`**`new_terminal`**(*columns: int*, *rows: int*, ***, *renderer: Optional[int] = None*, *tileset: Optional[tcod.tileset.Tileset] = None*, *vsync: bool = True*, *sdl_window_flags: Optional[int] = None*, *title: Optional[str] = None*) → tcod.context.Context

Create a new context with the desired console size.

Deprecated since version 11.16: `tcod.context.new` provides more options.

`tcod.context.`**`SDL_WINDOW_FULLSCREEN`**

Exclusive fullscreen mode.

It's generally not recommended to use this flag unless you know what you're doing. *SDL_WINDOW_FULLSCREEN_DESKTOP* should be used instead whenever possible.

`tcod.context.`**`SDL_WINDOW_FULLSCREEN_DESKTOP`**

A borderless fullscreen window at the desktop resolution.

`tcod.context.`**`SDL_WINDOW_HIDDEN`**

Window is hidden.

`tcod.context.`**`SDL_WINDOW_BORDERLESS`**

Window has no decorative border.

`tcod.context.`**`SDL_WINDOW_RESIZABLE`**

Window can be resized.

`tcod.context.`**`SDL_WINDOW_MINIMIZED`**

Window is minimized.

`tcod.context.`**`SDL_WINDOW_MAXIMIZED`**

Window is maximized.

`tcod.context.`**`SDL_WINDOW_INPUT_GRABBED`**
Window has grabbed the input.

`tcod.context.`**`SDL_WINDOW_ALLOW_HIGHDPI`**
High DPI mode, see the SDL documentation.

`tcod.context.`**`RENDERER_OPENGL`**
A renderer for older versions of OpenGL.

Should support OpenGL 1 and GLES 1

`tcod.context.`**`RENDERER_OPENGL2`**
An SDL2/OPENGL2 renderer. Usually faster than regular SDL2.

Recommended if you need a high performance renderer.

Should support OpenGL 2.0 and GLES 2.0.

`tcod.context.`**`RENDERER_SDL`**
Same as RENDERER_SDL2, but forces SDL2 into software mode.

`tcod.context.`**`RENDERER_SDL2`**
The main SDL2 renderer.

Rendering is decided by SDL2 and can be changed by using an SDL2 hint: https://wiki.libsdl.org/SDL_HINT_RENDER_DRIVER

CHAPTER 9

tcod.event - SDL2 Event Handling

A light-weight implementation of event handling built on calls to SDL.

Many event constants are derived directly from SDL. For example: `tcod.event.K_UP` and `tcod.event.SCANCODE_A` refer to SDL's `SDLK_UP` and `SDL_SCANCODE_A` respectfully. See this table for all of SDL's keyboard constants.

Printing any event will tell you its attributes in a human readable format. An events type attribute if omitted is just the classes name with all letters upper-case.

As a general guideline, you should use *`KeyboardEvent.sym`* for command inputs, and *`TextInput.text`* for name entry fields.

Remember to add the line `import tcod.event`, as importing this module is not implied by `import tcod`.

New in version 8.4.

class `tcod.event.`**Point**(*x*, *y*)

x
Alias for field number 0

y
Alias for field number 1

class `tcod.event.`**Event**(*type: Optional[str] = None*)
The base event class.

type
This events type.

Type str

sdl_event
When available, this holds a python-cffi 'SDL_Event*' pointer. All sub-classes have this attribute.

classmethod from_sdl_event(*sdl_event: Any*) → Any
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class `tcod.event.`**`Quit`**(*type: Optional[str] = None*)

An application quit request event.

For more info on when this event is triggered see: https://wiki.libsdl.org/SDL_EventType#SDL_QUIT

type

Always "QUIT".

Type str

classmethod from_sdl_event(*sdl_event: Any*) → tcod.event.Quit

Return a class instance from a python-cffi 'SDL_Event*' pointer.

class `tcod.event.`**`KeyboardEvent`**(*scancode: int*, *sym: int*, *mod: int*, *repeat: bool = False*)

type

Will be "KEYDOWN" or "KEYUP", depending on the event.

Type str

scancode

The keyboard scan-code, this is the physical location of the key on the keyboard rather than the keys symbol.

Type int

sym

The keyboard symbol.

Type int

mod

A bitmask of the currently held modifier keys.

You can use the following to check if a modifier key is held:

- ***tcod.event.KMOD_LSHIFT*** Left shift bit.
- ***tcod.event.KMOD_RSHIFT*** Right shift bit.
- ***tcod.event.KMOD_LCTRL*** Left control bit.
- ***tcod.event.KMOD_RCTRL*** Right control bit.
- ***tcod.event.KMOD_LALT*** Left alt bit.
- ***tcod.event.KMOD_RALT*** Right alt bit.
- ***tcod.event.KMOD_LGUI*** Left meta key bit.
- ***tcod.event.KMOD_RGUI*** Right meta key bit.
- ***tcod.event.KMOD_SHIFT*** `tcod.event.KMOD_LSHIFT | tcod.event.KMOD_RSHIFT`
- ***tcod.event.KMOD_CTRL*** `tcod.event.KMOD_LCTRL | tcod.event.KMOD_RCTRL`
- ***tcod.event.KMOD_ALT*** `tcod.event.KMOD_LALT | tcod.event.KMOD_RALT`
- ***tcod.event.KMOD_GUI*** `tcod.event.KMOD_LGUI | tcod.event.KMOD_RGUI`
- ***tcod.event.KMOD_NUM*** Num lock bit.
- ***tcod.event.KMOD_CAPS*** Caps lock bit.
- ***tcod.event.KMOD_MODE*** AltGr key bit.

For example, if shift is held then `event.mod & tcod.event.KMOD_SHIFT` will evaluate to a true value.

Type int

repeat
True if this event exists because of key repeat.

Type bool

classmethod from_sdl_event(*sdl_event: Any*) → Any
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class tcod.event.**KeyDown**(*scancode: int*, *sym: int*, *mod: int*, *repeat: bool = False*)

class tcod.event.**KeyUp**(*scancode: int*, *sym: int*, *mod: int*, *repeat: bool = False*)

class tcod.event.**MouseMotion**(*pixel: Tuple[int, int] = (0, 0)*, *pixel_motion: Tuple[int, int] = (0, 0)*, *tile: Tuple[int, int] = (0, 0)*, *tile_motion: Tuple[int, int] = (0, 0)*, *state: int = 0*)

type
Always "MOUSEMOTION".

Type str

pixel
The pixel coordinates of the mouse.

Type *Point*

pixel_motion
The pixel delta.

Type *Point*

tile
The integer tile coordinates of the mouse on the screen.

Type *Point*

tile_motion
The integer tile delta.

Type *Point*

state
A bitmask of which mouse buttons are currently held.

Will be a combination of the following names:

- tcod.event.BUTTON_LMASK
- tcod.event.BUTTON_MMASK
- tcod.event.BUTTON_RMASK
- tcod.event.BUTTON_X1MASK
- tcod.event.BUTTON_X2MASK

Type int

classmethod from_sdl_event(*sdl_event: Any*) → tcod.event.MouseMotion
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class `tcod.event.`**`MouseButtonEvent`**(*pixel: Tuple[int, int] = (0, 0), tile: Tuple[int, int] = (0, 0), button: int = 0*)

`type`
Will be "MOUSEBUTTONDOWN" or "MOUSEBUTTONUP", depending on the event.

Type str

`pixel`
The pixel coordinates of the mouse.

Type *Point*

`tile`
The integer tile coordinates of the mouse on the screen.

Type *Point*

`button`
Which mouse button.

This will be one of the following names:

- tcod.event.BUTTON_LEFT
- tcod.event.BUTTON_MIDDLE
- tcod.event.BUTTON_RIGHT
- tcod.event.BUTTON_X1
- tcod.event.BUTTON_X2

Type int

classmethod **`from_sdl_event`**(*sdl_event: Any*) → Any
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class `tcod.event.`**`MouseButtonDown`**(*pixel: Tuple[int, int] = (0, 0), tile: Tuple[int, int] = (0, 0), button: int = 0*)
Same as MouseButtonEvent but with `type="MouseButtonDown"`.

class `tcod.event.`**`MouseButtonUp`**(*pixel: Tuple[int, int] = (0, 0), tile: Tuple[int, int] = (0, 0), button: int = 0*)
Same as MouseButtonEvent but with `type="MouseButtonUp"`.

class `tcod.event.`**`MouseWheel`**(*x: int, y: int, flipped: bool = False*)

`type`
Always "MOUSEWHEEL".

Type str

`x`
Horizontal scrolling. A positive value means scrolling right.

Type int

`y`
Vertical scrolling. A positive value means scrolling away from the user.

Type int

flipped
If True then the values of *x* and *y* are the opposite of their usual values. This depends on the settings of the Operating System.

Type bool

classmethod from_sdl_event(*sdl_event: Any*) → tcod.event.MouseWheel
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class tcod.event.**TextInput**(*text: str*)

type
Always "TEXTINPUT".

Type str

text
A Unicode string with the input.

Type str

classmethod from_sdl_event(*sdl_event: Any*) → tcod.event.TextInput
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class tcod.event.**WindowEvent**(*type: Optional[str] = None*)

type
A window event could mean various event types.

Type str

classmethod from_sdl_event(*sdl_event: Any*) → Any
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class tcod.event.**WindowMoved**(*x: int*, *y: int*)

type
Always "WINDOWMOVED".

Type str

x
Movement on the x-axis.

Type int

x
Movement on the y-axis.

Type int

class tcod.event.**WindowResized**(*type: str*, *width: int*, *height: int*)

type
"WINDOWRESIZED" or "WINDOWSIZECHANGED"

Type str

width
The current width of the window.

Type int

height
The current height of the window.

Type int

class tcod.event.**Undefined**
This class is a place holder for SDL events without their own tcod.event class.

classmethod from_sdl_event(*sdl_event: Any*) → tcod.event.Undefined
Return a class instance from a python-cffi 'SDL_Event*' pointer.

class tcod.event.**EventDispatch**
This class dispatches events to methods depending on the events type attribute.

To use this class, make a sub-class and override the relevant *ev_** methods. Then send events to the dispatch method.

Changed in version 11.12: This is now a generic class. The type hists at the return value of `dispatch` and the *ev_** methods.

Example:

```
import tcod

MOVE_KEYS = {  # key_symbol: (x, y)
    # Arrow keys.
    tcod.event.K_LEFT: (-1, 0),
    tcod.event.K_RIGHT: (1, 0),
    tcod.event.K_UP: (0, -1),
    tcod.event.K_DOWN: (0, 1),
    tcod.event.K_HOME: (-1, -1),
    tcod.event.K_END: (-1, 1),
    tcod.event.K_PAGEUP: (1, -1),
    tcod.event.K_PAGEDOWN: (1, 1),
    tcod.event.K_PERIOD: (0, 0),
    # Numpad keys.
    tcod.event.K_KP_1: (-1, 1),
    tcod.event.K_KP_2: (0, 1),
    tcod.event.K_KP_3: (1, 1),
    tcod.event.K_KP_4: (-1, 0),
    tcod.event.K_KP_5: (0, 0),
    tcod.event.K_KP_6: (1, 0),
    tcod.event.K_KP_7: (-1, -1),
    tcod.event.K_KP_8: (0, -1),
    tcod.event.K_KP_9: (1, -1),
    tcod.event.K_CLEAR: (0, 0),  # Numpad `clear` key.
    # Vi Keys.
    tcod.event.K_h: (-1, 0),
    tcod.event.K_j: (0, 1),
    tcod.event.K_k: (0, -1),
    tcod.event.K_l: (1, 0),
    tcod.event.K_y: (-1, -1),
    tcod.event.K_u: (1, -1),
    tcod.event.K_b: (-1, 1),
    tcod.event.K_n: (1, 1),
}


class State(tcod.event.EventDispatch[None]):
    """A state-based superclass that converts `events` into `commands`.
```

(continues on next page)

(continued from previous page)

```
    The configuration used to convert events to commands are hard-coded
    in this example, but could be modified to be user controlled.

    Subclasses will override the `cmd_*` methods with their own
    functionality.  There could be a subclass for every individual state
    of your game.
    """

    def ev_quit(self, event: tcod.event.Quit) -> None:
        """The window close button was clicked or Alt+F$ was pressed."""
        print(event)
        self.cmd_quit()

    def ev_keydown(self, event: tcod.event.KeyDown) -> None:
        """A key was pressed."""
        print(event)
        if event.sym in MOVE_KEYS:
            # Send movement keys to the cmd_move method with parameters.
            self.cmd_move(*MOVE_KEYS[event.sym])
        elif event.sym == tcod.event.K_ESCAPE:
            self.cmd_escape()

    def ev_mousebuttondown(self, event: tcod.event.MouseButtonDown) -> None:
        """The window was clicked."""
        print(event)

    def ev_mousemotion(self, event: tcod.event.MouseMotion) -> None:
        """The mouse has moved within the window."""
        print(event)

    def cmd_move(self, x: int, y: int) -> None:
        """Intent to move: `x` and `y` is the direction, both may be 0."""
        print("Command move: " + str((x, y)))

    def cmd_escape(self) -> None:
        """Intent to exit this state."""
        print("Command escape.")
        self.cmd_quit()

    def cmd_quit(self) -> None:
        """Intent to exit the game."""
        print("Command quit.")
        raise SystemExit()


root_console = tcod.console_init_root(80, 60)
state = State()
while True:
    tcod.console_flush()
    for event in tcod.event.wait():
        state.dispatch(event)
```

dispatch(*event: Any*) → Optional[T]

Send an event to an *ev_** method.

* will be the *event.type* attribute converted to lower-case.

Values returned by *ev_** calls will be returned by this function. This value always defaults to None for any non-overridden method.

Changed in version 11.12: Now returns the return value of *ev_** methods. *event.type* values of None are deprecated.

ev_keydown(*event: tcod.event.KeyDown*) → Optional[T]
Called when a keyboard key is pressed or repeated.

ev_keyup(*event: tcod.event.KeyUp*) → Optional[T]
Called when a keyboard key is released.

ev_mousebuttondown(*event: tcod.event.MouseButtonDown*) → Optional[T]
Called when a mouse button is pressed.

ev_mousebuttonup(*event: tcod.event.MouseButtonUp*) → Optional[T]
Called when a mouse button is released.

ev_mousemotion(*event: tcod.event.MouseMotion*) → Optional[T]
Called when the mouse is moved.

ev_mousewheel(*event: tcod.event.MouseWheel*) → Optional[T]
Called when the mouse wheel is scrolled.

ev_quit(*event: tcod.event.Quit*) → Optional[T]
Called when the termination of the program is requested.

ev_textinput(*event: tcod.event.TextInput*) → Optional[T]
Called to handle Unicode input.

ev_windowclose(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window manager requests the window to be closed.

ev_windowenter(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window gains mouse focus.

ev_windowexposed(*event: tcod.event.WindowEvent*) → Optional[T]
Called when a window is exposed, and needs to be refreshed.

This usually means a call to `tcod.console_flush` is necessary.

ev_windowfocusgained(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window gains keyboard focus.

ev_windowfocuslost(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window loses keyboard focus.

ev_windowhidden(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window is hidden.

ev_windowleave(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window loses mouse focus.

ev_windowmaximized(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window is maximized.

ev_windowminimized(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window is minimized.

ev_windowmoved(*event: tcod.event.WindowMoved*) → Optional[T]
Called when the window is moved.

ev_windowresized(*event: tcod.event.WindowResized*) → Optional[T]
Called when the window is resized.

ev_windowrestored(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window is restored.

ev_windowshown(*event: tcod.event.WindowEvent*) → Optional[T]
Called when the window is shown.

ev_windowsizechanged(*event: tcod.event.WindowResized*) → Optional[T]
Called when the system or user changes the size of the window.

tcod.event.**get**() → Iterator[Any]
Return an iterator for all pending events.

Events are processed as the iterator is consumed. Breaking out of, or discarding the iterator will leave the remaining events on the event queue. It is also safe to call this function inside of a loop that is already handling events (the event iterator is reentrant.)

Example:

```
for event in tcod.event.get():
    if event.type == "QUIT":
        print(event)
        raise SystemExit()
    elif event.type == "KEYDOWN":
        print(event)
    elif event.type == "MOUSEBUTTONDOWN":
        print(event)
    elif event.type == "MOUSEMOTION":
        print(event)
    else:
        print(event)
# For loop exits after all current events are processed.
```

tcod.event.**wait**(*timeout: Optional[float] = None*) → Iterator[Any]
Block until events exist, then return an event iterator.

timeout is the maximum number of seconds to wait as a floating point number with millisecond precision, or it can be None to wait forever.

Returns the same iterator as a call to `tcod.event.get`.

Example:

```
for event in tcod.event.wait():
    if event.type == "QUIT":
        print(event)
        raise SystemExit()
    elif event.type == "KEYDOWN":
        print(event)
    elif event.type == "MOUSEBUTTONDOWN":
        print(event)
    elif event.type == "MOUSEMOTION":
        print(event)
    else:
        print(event)
# For loop exits on timeout or after at least one event is processed.
```

tcod.event.**get_mouse_state**() → tcod.event.MouseState
Return the current state of the mouse.

New in version 9.3.

CHAPTER 10

tcod.image - Image Handling

Functionality for handling images.

Python-tcod is unable to render pixels to the screen directly. If your image can't be represented as tiles then you'll need to use an alternative library for graphics rendering.

class `tcod.image.`**`Image`**(*width: int*, *height: int*)

Parameters

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

width

Read only width of this Image.

Type int

height

Read only height of this Image.

Type int

blit(*console: tcod.console.Console*, *x: float*, *y: float*, *bg_blend: int*, *scale_x: float*, *scale_y: float*, *angle: float*) → None

Blit onto a Console using scaling and rotation.

Parameters

- **console** (Console) – Blit destination Console.
- **x** (*float*) – Console X position for the center of the Image blit.
- **y** (*float*) – 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: tcod.console.Console*, *dest_x: int*, *dest_y: int*, *img_x: int = 0*, *img_y: int = 0*, *img_width: int = -1*, *img_height: int = -1*) → None

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: tcod.console.Console*, *x: int*, *y: int*, *width: int*, *height: int*, *bg_blend: int*) → None

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: Tuple[int, int, int]*) → None

Fill this entire Image with color.

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

classmethod from_array(*array: Any*) → tcod.image.Image

Create a new Image from a copy of an array-like object.

Example

```
>>> import numpy as np
>>> import tcod
>>> array = np.zeros((5, 5, 3), dtype=np.uint8)
>>> image = tcod.image.Image.from_array(array)
```

New in version 11.4.

get_alpha(*x: int*, *y: int*) → int

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: float*, *top: float*, *right: float*, *bottom: float*) → Tuple[int, int, int]

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** (*float*) – Left corner of the region.
- **top** (*float*) – Top corner of the region.
- **right** (*float*) – Right corner of the region.
- **bottom** (*float*) – 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: int*, *y: int*) → Tuple[int, int, int]

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() → None

Horizontally flip this Image.

invert() → None

Invert all colors in this Image.

put_pixel(*x: int, y: int, color: Tuple[int, int, int]*) → None

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: tcod.console.Console*) → None

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: int = 1*) → None

Rotate this Image clockwise in 90 degree steps.

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

save_as(*filename: str*) → None
Save the Image to a 32-bit .bmp or .png file.

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

scale(*width: int*, *height: int*) → None
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: Tuple[int, int, int]*) → None
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() → None
Vertically flip this Image.

tcod.image.**load**(*filename: str*) → numpy.ndarray
Load a PNG file as an RGBA array.

filename is the name of the file to load.

The returned array is in the shape: *(height, width, RGBA)*.

New in version 11.4.

CHAPTER 11

tcod.los - Line of Sight

This modules holds functions for NumPy-based line of sight algorithms.

`tcod.los.bresenham`(*start: Tuple[int, int], end: Tuple[int, int]*) → numpy.ndarray

Return a thin Bresenham line as a NumPy array of shape (length, 2).

start and *end* are the endpoints of the line. The result always includes both endpoints, and will always contain at least one index.

You might want to use the results as is, convert them into a list with `numpy.ndarray.tolist` or transpose them and use that to index another 2D array.

Example:

```
>>> import tcod
>>> tcod.los.bresenham((3, 5),(7, 7)).tolist()  # Convert into list.
[[3, 5], [4, 5], [5, 6], [6, 6], [7, 7]]
>>> tcod.los.bresenham((0, 0), (0, 0))
array([[0, 0]]...)
>>> tcod.los.bresenham((0, 0), (4, 4))[1:-1]  # Clip both endpoints.
array([[1, 1],
       [2, 2],
       [3, 3]]...)

>>> array = np.zeros((5, 5), dtype=np.int8)
>>> array
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=int8)
>>> tcod.los.bresenham((0, 0), (3, 4)).T  # Transposed results.
array([[0, 1, 1, 2, 3],
       [0, 1, 2, 3, 4]]...)
>>> indexes_ij = tuple(tcod.los.bresenham((0, 0), (3, 4)).T)
>>> array[indexes_ij] = np.arange(len(indexes_ij[0]))
```

(continues on next page)

(continued from previous page)

```
>>> array
array([[0, 0, 0, 0, 0],
       [0, 1, 2, 0, 0],
       [0, 0, 0, 3, 0],
       [0, 0, 0, 0, 4],
       [0, 0, 0, 0, 0]], dtype=int8)
>>> array[indexes_ij]
array([0, 1, 2, 3, 4], dtype=int8)
```

New in version 11.14.

CHAPTER 12

tcod.map - Field of View

libtcod map attributes and field-of-view functions.

class `tcod.map.`**`Map`**(*width: int, height: int, order: Union[typing_extensions.Literal['C'][C], typing_extensions.Literal['F'][F]] = 'C'*)

A map containing libtcod attributes.

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

Changed in version 4.3: Added *order* parameter.

Parameters

- **width** (*int*) – Width of the new Map.
- **height** (*int*) – Height of the new Map.
- **order** (*str*) – Which numpy memory order to use.

width

Read only width of this Map.

Type int

height

Read only height of this Map.

Type int

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'.

Example:

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

# 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]]...)

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

Deprecated since version 11.13: You no longer need to use this class to hold data for field-of-view or pathfinding as those functions can now take NumPy arrays directly. See `tcod.map.compute_fov` and `tcod.path`.

compute_fov(*x: int*, *y: int*, *radius: int = 0*, *light_walls: bool = True*, *algorithm: int = 12*) → None

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

If you already have transparency in a NumPy array then you could use `tcod.map_compute_fov` instead.

tcod.map.**compute_fov**(*transparency: numpy.ndarray, pov: Tuple[int, int], radius: int = 0, light_walls: bool = True, algorithm: int = 12*) → numpy.ndarray

Return a boolean mask of the area covered by a field-of-view.

transparency is a 2 dimensional array where all non-zero values are considered transparent. The returned array will match the shape of this array.

pov is the point-of-view origin point. Areas are visible if they can be seen from this position. *pov* should be a 2D index matching the axes of the *transparency* array, and must be within the bounds of the *transparency* array.

radius is the maximum view distance from *pov*. If this is zero then the maximum distance is used.

If *light_walls* is True then visible obstacles will be returned, otherwise only transparent areas will be.

algorithm is the field-of-view algorithm to run. The default value is *tcod.FOV_RESTRICTIVE*. The options are:

- *tcod.FOV_BASIC*: Simple ray-cast implementation.
- *tcod.FOV_DIAMOND*
- *tcod.FOV_SHADOW*: Recursive shadow caster.
- *tcod.FOV_PERMISSIVE(n)*: *n* starts at 0 (most restrictive) and goes up to 8 (most permissive.)
- *tcod.FOV_RESTRICTIVE*
- *tcod.FOV_SYMMETRIC_SHADOWCAST*

New in version 9.3.

Changed in version 11.0: The parameters *x* and *y* have been changed to *pov*.

Changed in version 11.17: Added *tcod.FOV_SYMMETRIC_SHADOWCAST* option.

Example

```
>>> explored = np.zeros((3, 5), dtype=bool, order="F")
>>> transparency = np.ones((3, 5), dtype=bool, order="F")
>>> transparency[:2, 2] = False
>>> transparency  # Transparent area.
array([[ True,  True, False,  True,  True],
       [ True,  True, False,  True,  True],
       [ True,  True,  True,  True,  True]]...)
>>> visible = tcod.map.compute_fov(transparency, (0, 0))
>>> visible  # Visible area.
array([[ True,  True,  True, False, False],
       [ True,  True,  True, False, False],
       [ True,  True,  True,  True, False]]...)
>>> explored |= visible  # Keep track of an explored area.
```

See also:

`numpy.where`: For selecting between two arrays using a boolean array, like the one returned by this function.

`numpy.select`: Select between arrays based on multiple conditions.

CHAPTER 13

tcod.noise - Noise Map Generators

Noise map generators are provided by this module.

The *Noise.sample_mgrid* and *Noise.sample_ogrid* methods perform much better than multiple calls to *Noise.get_point*.

Example:

```
import numpy as np
import tcod

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: int, algorithm: int = 2, implementation: int = 0, hurst: float = 0.5, lacunarity: float = 2.0, octaves: float = 4, seed: Optional[tcod.random.Random] = 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

A cffi pointer to a TCOD_noise_t object.

Type CData

__getitem__(*indexes: Any*) → numpy.ndarray

Sample a noise map through NumPy indexing.

This follows NumPy's advanced indexing rules, but allows for floating point values.

New in version 11.16.

get_point(*x: float = 0, y: float = 0, z: float = 0, w: float = 0*) → float

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: numpy.ndarray*) → numpy.ndarray

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: numpy.ndarray*) → numpy.ndarray

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 14

tcod.path - Pathfinding

This module provides a fast configurable pathfinding implementation.

To get started create a 2D NumPy array of integers where a value of zero is a blocked node and any higher value is the cost to move to that node. You then pass this array to `SimpleGraph`, and then pass that graph to `Pathfinder`.

Once you have a `Pathfinder` you call `Pathfinder.add_root` to set the root node. You can then get a path towards or away from the root with `Pathfinder.path_from` and `Pathfinder.path_to` respectively.

`SimpleGraph` includes a code example of the above process.

Changed in version 5.0: All path-finding functions now respect the NumPy array shape (if a NumPy array is used.)

class `tcod.path.`**`AStar`**(*cost: Any*, *diagonal: float = 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: int*, *start_y: int*, *goal_x: int*, *goal_y: int*) → List[Tuple[int, int]]

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.`**`CustomGraph`**(*shape: Tuple[int, ...], *, order: str = 'C'*)

A customizable graph defining how a pathfinder traverses the world.

If you only need to path over a 2D array with typical edge rules then you should use `SimpleGraph`. This is an advanced interface for defining custom edge rules which would allow things such as 3D movement.

The graph is created with a *shape* defining the size and number of dimensions of the graph. The *shape* can only be 4 dimensions or lower.

order determines what style of indexing the interface expects. This is inherited by the pathfinder and will affect the *ij/xy* indexing order of all methods in the graph and pathfinder objects. The default order of *"C"* is for *ij* indexing. The *order* can be set to *"F"* for *xy* indexing.

After this graph is created you'll need to add edges which define the rules of the pathfinder. These rules usually define movement in the cardinal and diagonal directions, but can also include stairway type edges. `set_heuristic` should also be called so that the pathfinder will use A*.

After all edge rules are added the graph can be used to make one or more `Pathfinder` instances.

Example:

```
>>> import numpy as np
>>> import tcod
>>> graph = tcod.path.CustomGraph((5, 5))
>>> cost = np.ones((5, 5), dtype=np.int8)
>>> CARDINAL = [
...     [0, 1, 0],
...     [1, 0, 1],
...     [0, 1, 0],
... ]
>>> graph.add_edges(edge_map=CARDINAL, cost=cost)
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.add_root((0, 0))
>>> pf.resolve()
>>> pf.distance
array([[0, 1, 2, 3, 4],
       [1, 2, 3, 4, 5],
       [2, 3, 4, 5, 6],
       [3, 4, 5, 6, 7],
       [4, 5, 6, 7, 8]]...)
>>> pf.path_to((3, 3))
array([[0, 0],
       [0, 1],
       [1, 1],
       [2, 1],
       [2, 2],
       [2, 3],
       [3, 3]]...)
```

New in version 11.13.

Changed in version 11.15: Added the *order* parameter.

add_edge(*edge_dir: Tuple[int, ...], edge_cost: int = 1, *, cost: numpy.ndarray, condition: Optional[numpy.ndarray] = None*) → None

Add a single edge rule.

edge_dir is a tuple with the same length as the graphs dimensions. The edge is relative to any node.

edge_cost is the cost multiplier of the edge. Its multiplied with the *cost* array to the edges actual cost.

cost is a NumPy array where each node has the cost for movement into that node. Zero or negative values are used to mark blocked areas.

condition is an optional array to mark which nodes have this edge. If the node in *condition* is zero then the edge will be skipped. This is useful to mark portals or stairs for some edges.

The expected indexing for *edge_dir*, *cost*, and *condition* depend on the graphs *order*.

Example:

```
>>> import numpy as np
>>> import tcod
>>> graph3d = tcod.path.CustomGraph((2, 5, 5))
>>> cost = np.ones((2, 5, 5), dtype=np.int8)
>>> up_stairs = np.zeros((2, 5, 5), dtype=np.int8)
>>> down_stairs = np.zeros((2, 5, 5), dtype=np.int8)
>>> up_stairs[0, 0, 4] = 1
>>> down_stairs[1, 0, 4] = 1
>>> CARDINAL = [[0, 1, 0], [1, 0, 1], [0, 1, 0]]
>>> graph3d.add_edges(edge_map=CARDINAL, cost=cost)
>>> graph3d.add_edge((1, 0, 0), 1, cost=cost, condition=up_stairs)
>>> graph3d.add_edge((-1, 0, 0), 1, cost=cost, condition=down_stairs)
>>> pf3d = tcod.path.Pathfinder(graph3d)
>>> pf3d.add_root((0, 1, 1))
>>> pf3d.path_to((1, 2, 2))
array([[0, 1, 1],
       [0, 1, 2],
       [0, 1, 3],
       [0, 0, 3],
       [0, 0, 4],
       [1, 0, 4],
       [1, 1, 4],
       [1, 1, 3],
       [1, 2, 3],
       [1, 2, 2]]...)
```

Note in the above example that both sets of up/down stairs were added, but bidirectional edges are not a requirement for the graph. One directional edges such as pits can be added which will only allow movement outwards from the root nodes of the pathfinder.

add_edges(*, *edge_map: Any*, *cost: numpy.ndarray*, *condition: Optional[numpy.ndarray] = None*) → None

Add a rule with multiple edges.

edge_map is a NumPy array mapping the edges and their costs. This is easier to understand by looking at the examples below. Edges are relative to center of the array. The center most value is always ignored. If *edge_map* has fewer dimensions than the graph then it will apply to the right-most axes of the graph.

cost is a NumPy array where each node has the cost for movement into that node. Zero or negative values are used to mark blocked areas.

condition is an optional array to mark which nodes have this edge. See `add_edge`. If *condition* is the same array as *cost* then the pathfinder will not move into open area from a non-open ones.

The expected indexing for *edge_map*, *cost*, and *condition* depend on the graphs *order*. You may need to transpose the examples below if you're using *xy* indexing.

Example:

```
# 2D edge maps:
CARDINAL = [  # Simple arrow-key moves.  Manhattan distance.
    [0, 1, 0],
    [1, 0, 1],
```

(continues on next page)

(continued from previous page)

```
    [0, 1, 0],
]
CHEBYSHEV = [  # Chess king moves.  Chebyshev distance.
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1],
]
EUCLIDEAN = [  # Approximate euclidean distance.
    [99, 70, 99],
    [70, 0, 70],
    [99, 70, 99],
]
EUCLIDEAN_SIMPLE = [  # Very approximate euclidean distance.
    [3, 2, 3],
    [2, 0, 2],
    [3, 2, 3],
]
KNIGHT_MOVE = [  # Chess knight L-moves.
    [0, 1, 0, 1, 0],
    [1, 0, 0, 0, 1],
    [0, 0, 0, 0, 0],
    [1, 0, 0, 0, 1],
    [0, 1, 0, 1, 0],
]
AXIAL = [  # https://www.redblobgames.com/grids/hexagons/
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 0],
]
# 3D edge maps:
CARDINAL_PLUS_Z = [  # Cardinal movement with Z up/down edges.
    [
        [0, 0, 0],
        [0, 1, 0],
        [0, 0, 0],
    ],
    [
        [0, 1, 0],
        [1, 0, 1],
        [0, 1, 0],
    ],
    [
        [0, 0, 0],
        [0, 1, 0],
        [0, 0, 0],
    ],
]
CHEBYSHEV_3D = [  # Chebyshev distance, but in 3D.
    [
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
    ],
    [
        [1, 1, 1],
        [1, 0, 1],
        [1, 1, 1],
```

(continues on next page)

(continued from previous page)

```
    ],
    [
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
    ],
]
```

set_heuristic(*, *cardinal: int = 0*, *diagonal: int = 0*, *z: int = 0*, *w: int = 0*) → None

Sets a pathfinder heuristic so that pathfinding can done with A*.

cardinal, *diagonal*, *z, and 'w* are the lower-bound cost of movement in those directions. Values above the lower-bound can be used to create a greedy heuristic, which will be faster at the cost of accuracy.

Example:

```
>>> import numpy as np
>>> import tcod
>>> graph = tcod.path.CustomGraph((5, 5))
>>> cost = np.ones((5, 5), dtype=np.int8)
>>> EUCLIDEAN = [[99, 70, 99], [70, 0, 70], [99, 70, 99]]
>>> graph.add_edges(edge_map=EUCLIDEAN, cost=cost)
>>> graph.set_heuristic(cardinal=70, diagonal=99)
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.add_root((0, 0))
>>> pf.path_to((4, 4))
array([[0, 0],
       [1, 1],
       [2, 2],
       [3, 3],
       [4, 4]]...)
>>> pf.distance
array([[         0,         70,        198, 2147483647, 2147483647],
       [        70,         99,        169,        297, 2147483647],
       [       198,        169,        198,        268,        396],
       [2147483647,        297,        268,        297,        367],
       [2147483647, 2147483647,        396,        367,        396]]...)
>>> pf.path_to((2, 0))
array([[0, 0],
       [1, 0],
       [2, 0]]...)
>>> pf.distance
array([[         0,         70,        198, 2147483647, 2147483647],
       [        70,         99,        169,        297, 2147483647],
       [       140,        169,        198,        268,        396],
       [       210,        239,        268,        297,        367],
       [2147483647, 2147483647,        396,        367,        396]]...)
```

Without a heuristic the above example would need to evaluate the entire array to reach the opposite side of it. With a heuristic several nodes can be skipped, which will process faster. Some of the distances in the above example look incorrect, that's because those nodes are only partially evaluated, but pathfinding to those nodes will work correctly as long as the heuristic isn't greedy.

ndim

The number of dimensions.

shape

The shape of this graph.

class tcod.path.**Dijkstra**(*cost: Any*, *diagonal: float = 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: int*, *y: int*) → List[Tuple[int, int]]
Return a list of (x, y) steps to reach the goal point, if possible.

set_goal(*x: int*, *y: int*) → None
Set the goal point and recompute the Dijkstra path-finder.

class tcod.path.**EdgeCostCallback**(*callback: Callable[[int, int, int, int], float], shape: Tuple[int, int]*)

Calculate cost from an edge-cost callback.

callback is the custom userdata to send to the C call.

shape is a 2-item tuple representing the maximum boundary for the algorithm. The callback will not be called with parameters outside of these bounds.

Changed in version 5.0: Now only accepts a *shape* argument instead of *width* and *height*.

class tcod.path.**NodeCostArray**

Calculate cost from a numpy array of nodes.

array is a NumPy array holding the path-cost of each node. A cost of 0 means the node is blocking.

static **__new__**(*cls*, *array: numpy.ndarray*) → tcod.path.NodeCostArray
Validate a numpy array and setup a C callback.

class tcod.path.**Pathfinder**(*graph: Union[tcod.path.CustomGraph, tcod.path.SimpleGraph]*)

A generic modular pathfinder.

How the pathfinder functions depends on the graph provided. see *SimpleGraph* for how to set one up.

New in version 11.13.

add_root(*index: Tuple[int, ...], value: int = 0*) → None
Add a root node and insert it into the pathfinder frontier.

index is the root point to insert. The length of *index* must match the dimensions of the graph.

value is the distance to use for this root. Zero is typical, but if multiple roots are added they can be given different weights.

clear() → None
Reset the pathfinder to its initial state.

This sets all values on the *distance* array to their maximum value.

path_from(*index: Tuple[int, ...]*) → numpy.ndarray
Return the shortest path from *index* to the nearest root.

The returned array is of shape *(length, ndim)* where *length* is the total inclusive length of the path and *ndim* is the dimensions of the pathfinder defined by the graph.

The return value is inclusive, including both the starting and ending points on the path. If the root point is unreachable or *index* is already at a root then *index* will be the only point returned.

This automatically calls *resolve* if the pathfinder has not yet reached *index*.

A common usage is to slice off the starting point and convert the array into a list.

Example:

```
>>> cost = np.ones((5, 5), dtype=np.int8)
>>> cost[:, 3:] = 0
>>> graph = tcod.path.SimpleGraph(cost=cost, cardinal=2, diagonal=3)
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.add_root((0, 0))
>>> pf.path_from((2, 2)).tolist()
[[2, 2], [1, 1], [0, 0]]
>>> pf.path_from((2, 2))[1:].tolist()  # Exclude the starting point by␣
↪slicing the array.
[[1, 1], [0, 0]]
>>> pf.path_from((4, 4)).tolist()  # Blocked paths will only have the index␣
↪point.
[[4, 4]]
>>> pf.path_from((4, 4))[1:].tolist()  # Exclude the starting point so that a␣
↪blocked path is an empty list.
[]
```

path_to(*index: Tuple[int, ...]*) → numpy.ndarray

Return the shortest path from the nearest root to *index*.

See `path_from`. This is an alias for `path_from(...)[::-1]`.

This is the method to call when the root is an entity to move to a position rather than a destination itself.

Example:

```
>>> graph = tcod.path.SimpleGraph(
...     cost=np.ones((5, 5), np.int8), cardinal=2, diagonal=3,
... )
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.add_root((0, 0))
>>> pf.path_to((0, 0)).tolist()  # This method always returns at least one␣
↪point.
[[0, 0]]
>>> pf.path_to((3, 3)).tolist()  # Always includes both ends on a valid path.
[[0, 0], [1, 1], [2, 2], [3, 3]]
>>> pf.path_to((3, 3))[1:].tolist()  # Exclude the starting point by slicing␣
↪the array.
[[1, 1], [2, 2], [3, 3]]
>>> pf.path_to((0, 0))[1:].tolist()  # Exclude the starting point so that a␣
↪blocked path is an empty list.
[]
```

rebuild_frontier() → None

Reconstruct the frontier using the current distance array.

If you are using `add_root` then you will not need to call this function. This is only needed if the `distance` array has been modified manually.

After you are finished editing `distance` you must call this function before calling `resolve` or any function which calls `resolve` implicitly such as `path_from` or `path_to`.

resolve(*goal: Optional[Tuple[int, ...]] = None*) → None

Manually run the pathfinder algorithm.

The `path_from` and `path_to` methods will automatically call this method on demand.

If *goal* is *None* then this will attempt to complete the entire `distance` and `traversal` arrays without a heuristic. This is similar to Dijkstra.

If *goal* is given an index then it will attempt to resolve the `distance` and `traversal` arrays only up to the *goal*. If the graph has set a heuristic then it will be used with a process similar to *A**.

Example:

```
>>> graph = tcod.path.SimpleGraph(
...     cost=np.ones((4, 4), np.int8), cardinal=2, diagonal=3,
... )
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.distance
array([[2147483647, 2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647]]...)
>>> pf.add_root((0, 0))
>>> pf.distance
array([[         0, 2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647]]...)
>>> pf.resolve((1, 1))  # Resolve up to (1, 1) as A*.
>>> pf.distance  # Partially resolved distance.
array([[         0,          2,          6, 2147483647],
       [         2,          3,          5, 2147483647],
       [         6,          5,          6, 2147483647],
       [2147483647, 2147483647, 2147483647, 2147483647]]...)
>>> pf.resolve()  # Resolve the full graph as Dijkstra.
>>> pf.distance  # Fully resolved distance.
array([[0, 2, 4, 6],
       [2, 3, 5, 7],
       [4, 5, 6, 8],
       [6, 7, 8, 9]]...)
```

`distance`

The distance values of the pathfinder.

The array returned from this property maintains the graphs *order*.

Unreachable or unresolved points will be at their maximum values. You can use `numpy.iinfo` if you need to check for these.

Example:

```
pf  # Resolved Pathfinder instance.
reachable = pf.distance != numpy.iinfo(pf.distance.dtype).max
reachable  # A boolean array of reachable area.
```

You may edit this array manually, but the pathfinder won't know of your changes until `rebuild_frontier` is called.

`traversal`

An array used to generate paths from any point to the nearest root.

The array returned from this property maintains the graphs *order*. It has an extra dimension which includes the index of the next path.

Example:

```
# This example demonstrates the purpose of the traversal array.
>>> graph = tcod.path.SimpleGraph(
```

(continues on next page)

(continued from previous page)

```
...     cost=np.ones((5, 5), np.int8), cardinal=2, diagonal=3,
... )
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.add_root((0, 0))
>>> pf.resolve()
>>> pf.traversal[3, 3].tolist()  # Faster.
[2, 2]
>>> pf.path_from((3, 3))[1].tolist()  # Slower.
[2, 2]
>>> i, j = (3, 3)  # Starting index.
>>> path = [(i, j)]  # List of nodes from the start to the root.
>>> while not (pf.traversal[i, j] == (i, j)).all():
...     i, j = pf.traversal[i, j]
...     path.append((i, j))
>>> path  # Slower.
[(3, 3), (2, 2), (1, 1), (0, 0)]
>>> pf.path_from((3, 3)).tolist()  # Faster.
[[3, 3], [2, 2], [1, 1], [0, 0]]
```

The above example is slow and will not detect infinite loops. Use `path_from` or `path_to` when you need to get a path.

As the pathfinder is resolved this array is filled

class `tcod.path.`**`SimpleGraph`**(*, *cost: numpy.ndarray*, *cardinal: int*, *diagonal: int*, *greed: int = 1*)

A simple 2D graph implementation.

cost is a NumPy array where each node has the cost for movement into that node. Zero or negative values are used to mark blocked areas. A reference of this array is used. Any changes to the array will be reflected in the graph.

cardinal and *diagonal* are the cost to move along the edges for those directions. The total cost to move from one node to another is the *cost* array value multiplied by the edge cost. A value of zero will block that direction.

greed is used to define the heuristic. To get the fastest accurate heuristic *greed* should be the lowest non-zero value on the *cost* array. Higher values may be used for an inaccurate but faster heuristic.

Example:

```
>>> import numpy as np
>>> import tcod
>>> cost = np.ones((5, 10), dtype=np.int8, order="F")
>>> graph = tcod.path.SimpleGraph(cost=cost, cardinal=2, diagonal=3)
>>> pf = tcod.path.Pathfinder(graph)
>>> pf.add_root((2, 4))
>>> pf.path_to((3, 7)).tolist()
[[2, 4], [2, 5], [2, 6], [3, 7]]
```

New in version 11.15.

`set_heuristic`(*, *cardinal: int*, *diagonal: int*) → None

Change the heuristic for this graph.

When created a `SimpleGraph` will automatically have a heuristic. So calling this method is often unnecessary.

cardinal and *diagonal* are weights for the heuristic. Higher values are more greedy. The default values are set to `cardinal * greed` and `diagonal * greed` when the `SimpleGraph` is created.

`tcod.path.dijkstra2d`(*distance: numpy.ndarray, cost: numpy.ndarray, cardinal: Optional[int] = None, diagonal: Optional[int] = None, *, edge_map: Any = None*) → None

Return the computed distance of all nodes on a 2D Dijkstra grid.

distance is an input/output array of node distances. Is this often an array filled with maximum finite values and 1 or more points with a low value such as 0. Distance will flow from these low values to adjacent nodes based the cost to reach those nodes. This array is modified in-place.

cost is an array of node costs. Any node with a cost less than or equal to 0 is considered blocked off. Positive values are the distance needed to reach that node.

cardinal and *diagonal* are the cost multipliers for edges in those directions. A value of None or 0 will disable those directions. Typical values could be: `1, None`, `1, 1`, `2, 3`, etc.

edge_map is a 2D array of edge costs with the origin point centered on the array. This can be used to define the edges used from one node to another. This parameter can be hard to understand so you should see how it's used in the examples.

Example:

```
>>> import numpy as np
>>> import tcod
>>> cost = np.ones((3, 3), dtype=np.uint8)
>>> cost[:2, 1] = 0
>>> cost
array([[1, 0, 1],
       [1, 0, 1],
       [1, 1, 1]], dtype=uint8)
>>> dist = tcod.path.maxarray((3, 3), dtype=np.int32)
>>> dist[0, 0] = 0
>>> dist
array([[         0, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647]]...)
>>> tcod.path.dijkstra2d(dist, cost, 2, 3)
>>> dist
array([[         0, 2147483647,         10],
       [         2, 2147483647,          8],
       [         4,          5,          7]]...)
>>> path = tcod.path.hillclimb2d(dist, (2, 2), True, True)
>>> path
array([[2, 2],
       [2, 1],
       [1, 0],
       [0, 0]], dtype=int32)
>>> path = path[::-1].tolist()
>>> while path:
...     print(path.pop(0))
[0, 0]
[1, 0]
[2, 1]
[2, 2]
```

edge_map is used for more complicated graphs. The following example uses a 'knight move' edge map.

Example:

```
>>> import numpy as np
>>> import tcod
>>> knight_moves = [
```

(continues on next page)

(continued from previous page)

```
...     [0, 1, 0, 1, 0],
...     [1, 0, 0, 0, 1],
...     [0, 0, 0, 0, 0],
...     [1, 0, 0, 0, 1],
...     [0, 1, 0, 1, 0],
... ]
>>> dist = tcod.path.maxarray((8, 8))
>>> dist[0,0] = 0
>>> cost = np.ones((8, 8), int)
>>> tcod.path.dijkstra2d(dist, cost, edge_map=knight_moves)
>>> dist
array([[0, 3, 2, 3, 2, 3, 4, 5],
       [3, 4, 1, 2, 3, 4, 3, 4],
       [2, 1, 4, 3, 2, 3, 4, 5],
       [3, 2, 3, 2, 3, 4, 3, 4],
       [2, 3, 2, 3, 4, 3, 4, 5],
       [3, 4, 3, 4, 3, 4, 5, 4],
       [4, 3, 4, 3, 4, 5, 4, 5],
       [5, 4, 5, 4, 5, 4, 5, 6]]...)
>>> tcod.path.hillclimb2d(dist, (7, 7), edge_map=knight_moves)
array([[7, 7],
       [5, 6],
       [3, 5],
       [1, 4],
       [0, 2],
       [2, 1],
       [0, 0]], dtype=int32)
```

edge_map can also be used to define a hex-grid. See https://www.redblobgames.com/grids/hexagons/ for more info. The following example is using axial coordinates.

Example:

```
hex_edges = [
    [0, 1, 1],
    [1, 0, 1],
    [1, 1, 0],
]
```

New in version 11.2.

Changed in version 11.13: Added the *edge_map* parameter.

`tcod.path.`**`hillclimb2d`**(*distance: numpy.ndarray, start: Tuple[int, int], cardinal: Optional[bool] = None, diagonal: Optional[bool] = None, *, edge_map: Any = None*) → numpy.ndarray

Return a path on a grid from *start* to the lowest point.

distance should be a fully computed distance array. This kind of array is returned by `dijkstra2d`.

start is a 2-item tuple with starting coordinates. The axes if these coordinates should match the axis of the *distance* array. An out-of-bounds *start* index will raise an IndexError.

At each step nodes adjacent toe current will be checked for a value lower than the current one. Which directions are checked is decided by the boolean values *cardinal* and *diagonal*. This process is repeated until all adjacent nodes are equal to or larger than the last point on the path.

If *edge_map* was used with `tcod.path.dijkstra2d` then it should be reused for this function. Keep in mind that *edge_map* must be bidirectional since hill-climbing will traverse the map backwards.

The returned array is a 2D NumPy array with the shape: (length, axis). This array always includes both the starting and ending point and will always have at least one item.

Typical uses of the returned array will be to either convert it into a list which can be popped from, or transpose it and convert it into a tuple which can be used to index other arrays using NumPy's advanced indexing rules.

New in version 11.2.

Changed in version 11.13: Added *edge_map* parameter.

`tcod.path.`**`maxarray`**(*shape: Tuple[int, ...], dtype: Any = <class 'numpy.int32'>, order: Union[typing_extensions.Literal['C'][C], typing_extensions.Literal['F'][F]] = 'C'*) → numpy.ndarray

Return a new array filled with the maximum finite value for *dtype*.

shape is of the new array. Same as other NumPy array initializers.

dtype should be a single NumPy integer type.

order can be "C" or "F".

This works the same as `np.full(shape, np.iinfo(dtype).max, dtype, order)`.

This kind of array is an ideal starting point for distance maps. Just set any point to a lower value such as 0 and then pass this array to a function such as `dijkstra2d`.

CHAPTER 15

tcod.random - Random Number Generators

Usually it's recommend to the Python's standard library *random* module instead of this one.

However, you will need to use these generators to get deterministic results from the `Noise` and `BSP` classes.

class `tcod.random.`**Random**(*algorithm: int = 0*, *seed: Optional[Hashable] = None*)

The libtcod random number generator.

algorithm defaults to Mersenne Twister, it can be one of:

- tcod.random.MERSENNE_TWISTER
- tcod.random.MULTIPLY_WITH_CARRY

seed is a 32-bit number or any Python hashable object like a string. Using the same seed will cause the generator to return deterministic values. The default *seed* of None will generate a random seed instead.

random_c

A cffi pointer to a TCOD_random_t object.

Type CData

Changed in version 9.1: Added *tcod.random.MULTIPLY_WITH_CARRY* constant. *algorithm* parameter now defaults to *tcod.random.MERSENNE_TWISTER*.

__getstate__() → Any

Pack the self.random_c attribute into a portable state.

__setstate__(*state: Any*) → None

Create a new cdata object with the stored paramaters.

guass(*mu: float*, *sigma: float*) → float

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: float*, *sigma: float*) → float

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: int*, *high: int*) → int

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: float*, *high: float*) → float

Return a random floating number in the range: low <= n <= high.

Parameters

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

Returns A random float.

Return type float

CHAPTER 16

tcod.tileset - Font Loading Functions

Tileset and font related functions.

Tilesets can be loaded as a whole from tile-sheets or True-Type fonts, or they can be put together from multiple tile images by loading them separately using `Tileset.set_tile`.

A major restriction with libtcod is that all tiles must be the same size and tiles can't overlap when rendered. For sprite-based rendering it can be useful to use an alternative library for graphics rendering while continuing to use python-tcod's pathfinding and field-of-view algorithms.

class `tcod.tileset.`**`Tileset`**(*tile_width: int*, *tile_height: int*)

A collection of graphical tiles.

This class is provisional, the API may change in the future.

`__contains__`(*codepoint: int*) → bool

Test if a tileset has a codepoint with `n in tileset`.

`get_tile`(*codepoint: int*) → numpy.ndarray

Return a copy of a tile for the given codepoint.

If the tile does not exist yet then a blank array will be returned.

The tile will have a shape of (height, width, rgba) and a dtype of uint8. Note that most grey-scale tiles will only use the alpha channel and will usually have a solid white color channel.

`remap`(*codepoint: int*, *x: int*, *y: int = 0*) → None

Reassign a codepoint to a character in this tileset.

codepoint is the Unicode codepoint to assign.

x and *y* is the position of the tilesheet to assign to *codepoint*. This is the tile position itself, not the pixel position of the tile. Large values of *x* will wrap to the next row, so using *x* by itself is equivalent to *Tile Index* in the *Character Table Reference*.

This is normally used on loaded tilesheets. Other methods of Tileset creation won't have reliable tile indexes.

New in version 11.12.

render(*console: tcod.console.Console*) → numpy.ndarray
Render an RGBA array, using console with this tileset.

console is the Console object to render, this can not be the root console.

The output array will be a np.uint8 array with the shape of: `(con_height * tile_height, con_width * tile_width, 4)`.

New in version 11.9.

set_tile(*codepoint: int*, *tile: numpy.ndarray*) → None
Upload a tile into this array.

The tile can be in 32-bit color (height, width, rgba), or grey-scale (height, width). The tile should have a dtype of `np.uint8`.

This data may need to be sent to graphics card memory, this is a slow operation.

tile_height
The height of the tile in pixels.

tile_shape
The shape (height, width) of the tile in pixels.

tile_width
The width of the tile in pixels.

tcod.tileset.**get_default**() → tcod.tileset.Tileset
Return a reference to the default Tileset.

New in version 11.10.

Deprecated since version 11.13: The default tileset is deprecated. With contexts this is no longer needed.

tcod.tileset.**load_bdf**(*path: str*) → tcod.tileset.Tileset
Return a new Tileset from a *.bdf* file.

For the best results the font should be monospace, cell-based, and single-width. As an example, a good set of fonts would be the Unicode fonts and tools for X11 package.

Pass the returned Tileset to `tcod.tileset.set_default` and it will take effect when *tcod.console_init_root* is called.

New in version 11.10.

tcod.tileset.**load_tilesheet**(*path: str, columns: int, rows: int, charmap: Optional[Iterable[int]]*) → tcod.tileset.Tileset
Return a new Tileset from a simple tilesheet image.

path is the file path to a PNG file with the tileset.

columns and *rows* is the shape of the tileset. Tiles are assumed to take up the entire space of the image.

charmap is the character mapping to use. This is a list or generator of codepoints which map the tiles like this: `charmap[tile_index] = codepoint`. For common tilesets *charmap* should be `tcod.tileset.CHARMAP_CP437`. Generators will be sliced so `itertools.count` can be used which will give all tiles the same codepoint as their index, but this will not map tiles onto proper Unicode. If *None* is used then no tiles will be mapped, you will need to use `Tileset.remap` to assign codepoints to this Tileset.

New in version 11.12.

tcod.tileset.**load_truetype_font**(*path: str*, *tile_width: int*, *tile_height: int*) → tcod.tileset.Tileset
Return a new Tileset from a *.ttf* or *.otf* file.

Same as `set_truetype_font`, but returns a `Tileset` instead. You can send this Tileset to `set_default`.

This function is provisional. The API may change.

`tcod.tileset.`**`set_default`**(*tileset: tcod.tileset.Tileset*) → None

Set the default tileset.

The display will use this new tileset immediately.

New in version 11.10.

Deprecated since version 11.13: The default tileset is deprecated. With contexts this is no longer needed.

`tcod.tileset.`**`set_truetype_font`**(*path: str*, *tile_width: int*, *tile_height: int*) → None

Set the default tileset from a *.ttf* or *.otf* file.

path is the file path for the font file.

tile_width and *tile_height* are the desired size of the tiles in the new tileset. The font will be scaled to fit the given *tile_height* and *tile_width*.

This function must be called before `tcod.console_init_root`. Once the root console is setup you may call this funtion again to change the font. The tileset can be changed but the window will not be resized automatically.

New in version 9.2.

Deprecated since version 11.13: This function does not support contexts. Use `load_truetype_font` instead.

`tcod.tileset.`**`CHARMAP_CP437 = [0, 9786, 9787, 9829, 9830, 9827, 9824, 8226, 9688, 9675, 968`**

A code page 437 character mapping.

See *Code Page 437* for more info and a table of glyphs.

New in version 11.12.

`tcod.tileset.`**`CHARMAP_TCOD = [32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 4`**

The layout used by older libtcod fonts, in Unicode.

This layout is non-standard, and it's not recommend to make a font for it, but you might need it to load an existing font made for libtcod.

This character map is in Unicode, so old code using the non-Unicode *tcod.CHAR_** constants will need to be updated.

See *Deprecated TCOD Layout* for a table of glyphs used in this character map.

New in version 11.12.

CHAPTER 17

libtcodpy - Old API Functions

This is all the functions included since the start of the Python port. This collection is often called *libtcodpy*, the name of the original Python port. These functions are reproduced by python-tcod in their entirely.

A large majority of these functions are deprecated and will be removed in the future. In general this entire section should be avoided whenever possible. See *Getting Started* for how to make a new python-tcod project with its modern API.

17.1 bsp

`tcod.bsp_new_with_size`(*x: int*, *y: int*, *w: int*, *h: int*) → tcod.bsp.BSP

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: tcod.bsp.BSP*, *horizontal: bool*, *position: int*) → None

Deprecated since version 2.0: Use `BSP.split_once` instead.

`tcod.bsp_split_recursive`(*node: tcod.bsp.BSP, randomizer: Optional[tcod.random.Random], nb: int, minHSize: int, minVSize: int, maxHRatio: int, maxVRatio: int*) → None

Deprecated since version 2.0: Use `BSP.split_recursive` instead.

`tcod.bsp_resize`(*node: tcod.bsp.BSP*, *x: int*, *y: int*, *w: int*, *h: int*) → None
Deprecated since version 2.0: Assign directly to `BSP` attributes instead.

`tcod.bsp_left`(*node: tcod.bsp.BSP*) → Optional[tcod.bsp.BSP]
Deprecated since version 2.0: Use `BSP.children` instead.

`tcod.bsp_right`(*node: tcod.bsp.BSP*) → Optional[tcod.bsp.BSP]
Deprecated since version 2.0: Use `BSP.children` instead.

`tcod.bsp_father`(*node: tcod.bsp.BSP*) → Optional[tcod.bsp.BSP]
Deprecated since version 2.0: Use `BSP.parent` instead.

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

`tcod.bsp_contains`(*node: tcod.bsp.BSP*, *cx: int*, *cy: int*) → bool
Deprecated since version 2.0: Use `BSP.contains` instead.

`tcod.bsp_find_node`(*node: tcod.bsp.BSP*, *cx: int*, *cy: int*) → Optional[tcod.bsp.BSP]
Deprecated since version 2.0: Use `BSP.find_node` instead.

`tcod.bsp_traverse_pre_order`(*node: tcod.bsp.BSP, callback: Callable[[tcod.bsp.BSP, Any], None], userData: Any = 0*) → None
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use `BSP.pre_order` instead.

`tcod.bsp_traverse_in_order`(*node: tcod.bsp.BSP, callback: Callable[[tcod.bsp.BSP, Any], None], userData: Any = 0*) → None
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use `BSP.in_order` instead.

`tcod.bsp_traverse_post_order`(*node: tcod.bsp.BSP, callback: Callable[[tcod.bsp.BSP, Any], None], userData: Any = 0*) → None
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use `BSP.post_order` instead.

`tcod.bsp_traverse_level_order`(*node: tcod.bsp.BSP, callback: Callable[[tcod.bsp.BSP, Any], None], userData: Any = 0*) → None
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use `BSP.level_order` instead.

`tcod.bsp_traverse_inverted_level_order`(*node: tcod.bsp.BSP, callback: Callable[[tcod.bsp.BSP, Any], None], userData: Any = 0*) → None
Traverse this nodes hierarchy with a callback.

Deprecated since version 2.0: Use `BSP.inverted_level_order` instead.

`tcod.bsp_remove_sons`(*node: tcod.bsp.BSP*) → None
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: tcod.bsp.BSP*) → None
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.

17.2 color

class `tcod.Color`(*r: int = 0, g: int = 0, b: int = 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

Red value, always normalised to 0-255.

Deprecated since version 9.2: Color attributes will not be mutable in the future.

Type int

g

Green value, always normalised to 0-255.

Deprecated since version 9.2: Color attributes will not be mutable in the future.

Type int

b

Blue value, always normalised to 0-255.

Deprecated since version 9.2: Color attributes will not be mutable in the future.

Type int

__getitem__(*index: Any*) → Any

Deprecated since version 9.2: Accessing colors via a letter index is deprecated.

__eq__(*other: Any*) → bool

Compare equality between colors.

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

__repr__() → str

Return a printable representation of the current color.

`tcod.`**color_lerp**(*c1: Tuple[int, int, int], c2: Tuple[int, int, int], a: float*) → tcod.color.Color

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: tcod.color.Color*, *h: float*, *s: float*, *v: float*) → None

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: Tuple[int, int, int]*) → Tuple[float, float, float]

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: tcod.color.Color*, *scoef: float*, *vcoef: float*) → None

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: Iterable[Tuple[int, int, int]]*, *indexes: Iterable[int]*) → List[tcod.color.Color]

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)]
```

17.2.1 color controls

Configurable color control constants which can be set up with `tcod.console_set_color_control`.

`tcod.COLCTRL_1`

`tcod.COLCTRL_2`

`tcod.COLCTRL_3`

`tcod.COLCTRL_4`

`tcod.COLCTRL_5`

`tcod.COLCTRL_STOP`

`tcod.COLCTRL_FORE_RGB`

`tcod.COLCTRL_BACK_RGB`

17.3 console

`tcod.console_set_custom_font`(*fontFile: AnyStr, flags: int = 1, nb_char_horiz: int = 0, nb_char_vertic: int = 0*) → None

Load the custom font file at *fontFile*.

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

Flags can be a mix of the following:

- tcod.FONT_LAYOUT_ASCII_INCOL: Decode tileset raw in column-major order.
- tcod.FONT_LAYOUT_ASCII_INROW: Decode tileset raw in row-major order.
- tcod.FONT_TYPE_GREYSCALE: Force tileset to be read as greyscale.
- tcod.FONT_TYPE_GRAYSCALE
- tcod.FONT_LAYOUT_TCOD: Unique layout used by libtcod.
- tcod.FONT_LAYOUT_CP437: Decode a row-major Code Page 437 tileset into Unicode.

nb_char_horiz and *nb_char_vertic* are the columns and rows of the font file respectfully.

Deprecated since version 11.13: Load fonts using `tcod.tileset.load_tileheet` instead. See *Getting Started* for more info.

`tcod.console_init_root`(*w: int, h: int, title: Optional[str] = None, fullscreen: bool = False, renderer: Optional[int] = None, order: Union[typing_extensions.Literal['C'][C], typing_extensions.Literal['F'][F]] = 'C', vsync: Optional[bool] = None*) → tcod.console.Console

Set up the primary display and return the root console.

w and *h* are the columns and rows of the new window (in tiles.)

title is an optional string to display on the windows title bar.

fullscreen determines if the window will start in fullscreen. Fullscreen mode is unreliable unless the renderer is set to *tcod.RENDERER_SDL2* or *tcod.RENDERER_OPENGL2*.

renderer is the rendering back-end that libtcod will use. If you don't know which to pick, then use *tcod.RENDERER_SDL2*. Options are:

- *tcod.RENDERER_SDL*: Forces the SDL2 renderer into software mode.

- *tcod.RENDERER_OPENGL*: An OpenGL 1 implementation.
- *tcod.RENDERER_GLSL*: A deprecated SDL2/OpenGL2 renderer.
- *tcod.RENDERER_SDL2*: The recommended SDL2 renderer. Rendering is decided by SDL2 and can be changed by using an SDL2 hint.
- *tcod.RENDERER_OPENGL2*: An SDL2/OPENGL2 renderer. Usually faster than regular SDL2. Requires OpenGL 2.0 Core.

order will affect how the array attributes of the returned root console are indexed. *order='C'* is the default, but *order='F'* is recommended.

If *vsync* is True then the frame-rate will be synchronized to the monitors vertical refresh rate. This prevents screen tearing and avoids wasting computing power on overdraw. If *vsync* is False then the frame-rate will be uncapped. The default is False but will change to True in the future. This option only works with the SDL2 or OPENGL2 renderers, any other renderer will always have *vsync* disabled.

The returned object is the root console. You don't need to use this object but you should at least close it when you're done with the libtcod window. You can do this by calling `Console.close` or by using this function in a context, like in the following example:

```
with tcod.console_init_root(80, 50, vsync=True) as root_console:
    ...  # Put your game loop here.
...  # Window closes at the end of the above block.
```

Changed in version 4.3: Added *order* parameter. *title* parameter is now optional.

Changed in version 8.0: The default *renderer* is now automatic instead of always being *RENDERER_SDL*.

Changed in version 10.1: Added the *vsync* parameter.

Deprecated since version 11.13: Use `tcod.context` for window management. See *Getting Started* for more info.

`tcod.console_flush`(*console: Optional[tcod.console.Console] = None, *, keep_aspect: bool = False, integer_scaling: bool = False, snap_to_integer: Optional[bool] = None, clear_color: Union[Tuple[int, int, int], Tuple[int, int, int, int]] = (0, 0, 0), align: Tuple[float, float] = (0.5, 0.5)*) → None

Update the display to represent the root consoles current state.

console is the console you want to present. If not given the root console will be used.

If *keep_aspect* is True then the console aspect will be preserved with a letterbox. Otherwise the console will be stretched to fill the screen.

If *integer_scaling* is True then the console will be scaled in integer increments. This will have no effect if the console must be shrunk. You can use `tcod.console.recommended_size` to create a console which will fit the window without needing to be scaled.

clear_color is an RGB or RGBA tuple used to clear the screen before the console is presented, this will normally affect the border/letterbox color.

align determines where the console will be placed when letter-boxing exists. Values of 0 will put the console at the upper-left corner. Values of 0.5 will center the console.

snap_to_integer is deprecated and setting it will have no effect. It will be removed in a later version.

Changed in version 11.8: The parameters *console*, *keep_aspect*, *integer_scaling*, *snap_to_integer*, *clear_color*, and *align* were added.

Changed in version 11.11: *clear_color* can now be an RGB tuple.

See also:

`tcod.console_init_root` `tcod.console.recommended_size`

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_blit`(*src: tcod.console.Console*, *x: int*, *y: int*, *w: int*, *h: int*, *dst: tcod.console.Console*, *xdst: int*, *ydst: int*, *ffade: float = 1.0*, *bfade: float = 1.0*) → None

Blit the console src from x,y,w,h to console dst at xdst,ydst.

Deprecated since version 8.5: Call the `Console.blit` method instead.

`tcod.console_check_for_keypress`(*flags: int = 2*) → tcod.libtcodpy.Key

Deprecated since version 9.3: Use the `tcod.event.get` function to check for events.

`tcod.console_clear`(*con: tcod.console.Console*) → None

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`

Deprecated since version 8.5: Call the `Console.clear` method instead.

`tcod.console_credits`() → None

`tcod.console_credits_render`(*x: int*, *y: int*, *alpha: bool*) → bool

`tcod.console_credits_reset`() → None

`tcod.console_delete`(*con: tcod.console.Console*) → None

Closes the window if *con* is the root console.

libtcod objects are automatically garbage collected once they go out of scope.

This function exists for backwards compatibility.

Deprecated since version 9.3: This function is not needed for normal `tcod.console.Console`'s. The root console should be used in a with statement instead to ensure that it closes.

`tcod.console_fill_background`(*con: tcod.console.Console, r: Sequence[int], g: Sequence[int], b: Sequence[int]*) → None

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.

Deprecated since version 8.4: You should assign to `tcod.console.Console.bg` instead.

`tcod.console_fill_char`(*con: tcod.console.Console, arr: Sequence[int]*) → None

Fill the character tiles of a console with an array.

arr is an array of integers with a length of the consoles width and height.

Deprecated since version 8.4: You should assign to `tcod.console.Console.ch` instead.

`tcod.console_fill_foreground`(*con: tcod.console.Console, r: Sequence[int], g: Sequence[int], b: Sequence[int]*) → None

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.

Deprecated since version 8.4: You should assign to *tcod.console.Console.fg* instead.

tcod.**console_from_file**(*filename: str*) → tcod.console.Console
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: str*) → tcod.console.Console
Return a single console from a REXPaint *.xp* file.

tcod.**console_get_alignment**(*con: tcod.console.Console*) → int
Return this consoles current alignment mode.

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

Deprecated since version 8.5: Check *Console.default_alignment* instead.

tcod.**console_get_background_flag**(*con: tcod.console.Console*) → int
Return this consoles current blend mode.

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

Deprecated since version 8.5: Check *Console.default_bg_blend* instead.

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

Deprecated since version 8.4: Array access performs significantly faster than using this function. See *Console.ch*.

tcod.**console_get_char_background**(*con: tcod.console.Console*, *x: int*, *y: int*) → tcod.color.Color
Return the background color at the x,y of this console.

Deprecated since version 8.4: Array access performs significantly faster than using this function. See *Console.bg*.

tcod.**console_get_char_foreground**(*con: tcod.console.Console*, *x: int*, *y: int*) → tcod.color.Color
Return the foreground color at the x,y of this console.

Deprecated since version 8.4: Array access performs significantly faster than using this function. See *Console.fg*.

tcod.**console_get_default_background**(*con: tcod.console.Console*) → tcod.color.Color
Return this consoles default background color.

Deprecated since version 8.5: Use *Console.default_bg* instead.

tcod.**console_get_default_foreground**(*con: tcod.console.Console*) → tcod.color.Color
Return this consoles default foreground color.

Deprecated since version 8.5: Use *Console.default_fg* instead.

tcod.**console_get_fade**() → int
Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_get_fading_color()` → tcod.color.Color

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_get_height(con: tcod.console.Console)` → int

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.height* instead.

`tcod.console_get_height_rect(con: tcod.console.Console, x: int, y: int, w: int, h: int, fmt: str)` → int

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

Deprecated since version 8.5: Use `Console.get_height_rect` instead.

`tcod.console_get_width(con: tcod.console.Console)` → int

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.width* instead.

`tcod.console_hline(con: tcod.console.Console, x: int, y: int, l: int, flag: int = 13)` → None

Draw a horizontal line on the console.

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

Deprecated since version 8.5: Use `Console.hline` instead.

`tcod.console_is_fullscreen()` → bool

Returns True if the display is fullscreen.

Returns True if the display is fullscreen, otherwise False.

Return type bool

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_is_key_pressed(key: int)` → bool

`tcod.console_is_window_closed()` → bool

Returns True if the window has received and exit event.

Deprecated since version 9.3: Use the `tcod.event` module to check for "QUIT" type events.

`tcod.console_load_apf(con: tcod.console.Console, filename: str)` → bool

Update a console from an ASCII Paint *.apf* file.

`tcod.console_load_asc(con: tcod.console.Console, filename: str)` → bool

Update a console from a non-delimited ASCII *.asc* file.

`tcod.console_load_xp(con: tcod.console.Console, filename: str)` → bool

Update a console from a REXPaint *.xp* file.

Deprecated since version 11.18: Functions modifying console objects in-place are deprecated. Use `tcod.console_from_xp` to load a Console from a file.

`tcod.console_list_load_xp`(*filename: str*) → Optional[List[tcod.console.Console]]
Return a list of consoles from a REXPaint *.xp* file.

`tcod.console_list_save_xp`(*console_list: Sequence[tcod.console.Console], filename: str, compress_level: int = 9*) → bool
Save a list of consoles to a REXPaint *.xp* file.

`tcod.console_map_ascii_code_to_font`(*asciiCode: int, fontCharX: int, fontCharY: int*) → None
Set a character code to new coordinates on the tile-set.

asciiCode should be any Unicode codepoint.

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.

Deprecated since version 11.13: Setup fonts using the `tcod.tileset` module. `Tileset.remap` replaces this function.

`tcod.console_map_ascii_codes_to_font`(*firstAsciiCode: int, nbCodes: int, fontCharX: int, fontCharY: int*) → None
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.

Deprecated since version 11.13: Setup fonts using the `tcod.tileset` module. `Tileset.remap` replaces this function.

`tcod.console_map_string_to_font`(*s: str, fontCharX: int, fontCharY: int*) → None
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 *'x00'* 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.

Deprecated since version 11.13: Setup fonts using the `tcod.tileset` module. `Tileset.remap` replaces this function.

`tcod.console_new`(*w: int*, *h: int*) → tcod.console.Console

Return an offscreen console of size: w,h.

Deprecated since version 8.5: Create new consoles using `tcod.console.Console` instead of this function.

`tcod.console_print`(*con: tcod.console.Console*, *x: int*, *y: int*, *fmt: str*) → None

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.

Deprecated since version 8.5: Use `Console.print_` instead.

`tcod.console_print_ex`(*con: tcod.console.Console*, *x: int*, *y: int*, *flag: int*, *alignment: int*, *fmt: str*) → None

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.

Deprecated since version 8.5: Use `Console.print_` instead.

`tcod.console_print_frame`(*con: tcod.console.Console*, *x: int*, *y: int*, *w: int*, *h: int*, *clear: bool = True*, *flag: int = 13*, *fmt: str = ''*) → None

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. If *fmt* is empty then no title will be drawn.

Changed in version 8.2: Now supports Unicode strings.

Deprecated since version 8.5: Use `Console.print_frame` instead.

`tcod.console_print_rect`(*con: tcod.console.Console*, *x: int*, *y: int*, *w: int*, *h: int*, *fmt: str*) → int

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

Deprecated since version 8.5: Use `Console.print_rect` instead.

`tcod.console_print_rect_ex`(*con: tcod.console.Console*, *x: int*, *y: int*, *w: int*, *h: int*, *flag: int*, *alignment: int*, *fmt: str*) → int

Print a string constrained to a rectangle with blend and alignment.

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

Return type int

Deprecated since version 8.5: Use `Console.print_rect` instead.

`tcod.console_put_char`(*con: tcod.console.Console, x: int, y: int, c: Union[int, str], flag: int = 13*) → None

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: tcod.console.Console, x: int, y: int, c: Union[int, str], fore: Tuple[int, int, int], back: Tuple[int, int, int]*) → None

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: tcod.console.Console, x: int, y: int, w: int, h: int, clr: bool, flag: int = 13*) → None

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

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

Deprecated since version 8.5: Use *Console.rect* instead.

`tcod.console_save_apf`(*con: tcod.console.Console, filename: str*) → bool

Save a console to an ASCII Paint *.apf* file.

`tcod.console_save_asc`(*con: tcod.console.Console, filename: str*) → bool

Save a console to a non-delimited ASCII *.asc* file.

`tcod.console_save_xp`(*con: tcod.console.Console, filename: str, compress_level: int = 9*) → bool

Save a console to a REXPaint *.xp* file.

`tcod.console_set_alignment`(*con: tcod.console.Console, alignment: int*) → None

Change this consoles current alignment mode.

- tcod.LEFT
- tcod.CENTER
- tcod.RIGHT

Parameters

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

Deprecated since version 8.5: Set `Console.default_alignment` instead.

`tcod.console_set_background_flag`(*con: tcod.console.Console, flag: int*) → None

Change the default blend mode for this console.

Parameters

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

Deprecated since version 8.5: Set `Console.default_bg_blend` instead.

`tcod.console_set_char`(*con: tcod.console.Console, x: int, y: int, c: Union[int, str]*) → None

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.

Deprecated since version 8.4: Array access performs significantly faster than using this function. See `Console.ch`.

`tcod.console_set_char_background`(*con: tcod.console.Console, x: int, y: int, col: Tuple[int, int, int], flag: int = 1*) → None

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: tcod.console.Console, x: int, y: int, col: Tuple[int, int, int]*) → None

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.

Deprecated since version 8.4: Array access performs significantly faster than using this function. See `Console.fg`.

`tcod.console_set_color_control`(*con: int, fore: Tuple[int, int, int], back: Tuple[int, int, int]*) → None

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: tcod.console.Console, col: Tuple[int, int, int]*) → None

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.

Deprecated since version 8.5: Use *Console.default_bg* instead.

`tcod.console_set_default_foreground`(*con: tcod.console.Console, col: Tuple[int, int, int]*) → None

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.

Deprecated since version 8.5: Use *Console.default_fg* instead.

`tcod.console_set_fade`(*fade: int, fadingColor: Tuple[int, int, int]*) → None

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_set_fullscreen`(*fullscreen: bool*) → None

Change the display to be fullscreen or windowed.

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

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_set_key_color`(*con: tcod.console.Console, col: Tuple[int, int, int]*) → None

Set a consoles blit transparent color.

Deprecated since version 8.5: Pass the key color to *tcod.console.Console.blit* instead of calling this function.

`tcod.console_set_window_title`(*title: str*) → None

Change the current title bar string.

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

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.console_vline`(*con: tcod.console.Console, x: int, y: int, l: int, flag: int = 13*) → None

Draw a vertical line on the console.

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

Deprecated since version 8.5: Use *Console.vline* instead.

`tcod.console_wait_for_keypress`(*flush: bool*) → tcod.libtcodpy.Key

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*

Deprecated since version 9.3: Use the `tcod.event.wait` function to wait for events.

17.4 Event

class tcod.**Key**

Key Event instance

vk

TCOD_keycode_t key code

Type int

c

character if vk == TCODK_CHAR else 0

Type int

text

text[TCOD_KEY_TEXT_SIZE]; text if vk == TCODK_TEXT else text[0] == ''

Type Text

pressed

does this correspond to a key press or key release event?

Type bool

lalt

True when left alt is held.

Type bool

lctrl

True when left control is held.

Type bool

lmeta

True when left meta key is held.

Type bool

ralt

True when right alt is held.

Type bool

rctrl

True when right control is held.

Type bool

rmeta

True when right meta key is held.

Type bool

shift
True when any shift is held.

Type bool

Deprecated since version 9.3: Use events from the `tcod.event` module instead.

__repr__() → str
Return a representation of this Key object.

class tcod.**Mouse**
Mouse event instance

x
Absolute mouse position at pixel x.

Type int

y

Type int

dx
Movement since last update in pixels.

Type int

dy

Type int

cx
Cell coordinates in the root console.

Type int

cy

Type int

dcx
Movement since last update in console cells.

Type int

dcy

Type int

lbutton
Left button status.

Type bool

rbutton
Right button status.

Type bool

mbutton
Middle button status.

Type bool

lbutton_pressed
Left button pressed event.

Type bool

rbutton_pressed
Right button pressed event.

Type bool

mbutton_pressed
Middle button pressed event.

Type bool

wheel_up
Wheel up event.

Type bool

wheel_down
Wheel down event.

Type bool

Deprecated since version 9.3: Use events from the `tcod.event` module instead.

__repr__() → str
Return a representation of this Mouse object.

17.4.1 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`

17.5 sys

`tcod.sys_set_fps`(*fps: int*) → None

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)

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_get_fps`() → int

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

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_get_last_frame_length`() → float

Return the delta time of the last rendered frame in seconds.

Returns The delta time of the last rendered frame.

Return type float

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_sleep_milli`(*val: int*) → None

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`() → int

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`() → float

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: int*) → None

Change the current rendering mode to renderer.

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_get_renderer`() → int

Return the current rendering mode.

Deprecated since version 11.13: This function is not supported by contexts. Check `Context.renderer_type` instead.

`tcod.sys_save_screenshot`(*name: Optional[str] = None*) → 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.

Deprecated since version 11.13: This function is not supported by contexts. Use `Context.save_screenshot` instead.

`tcod.sys_force_fullscreen_resolution`(*width: int*, *height: int*) → None

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.

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_get_current_resolution`() → Tuple[int, int]

Return a monitors pixel resolution as (width, height).

Deprecated since version 11.13: This function is deprecated, which monitor is detected is ambiguous.

`tcod.sys_get_char_size`() → Tuple[int, int]

Return the current fonts character size as (width, height)

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

Return type Tuple[int,int]

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_update_char`(*asciiCode: int*, *fontx: int*, *fonty: int*, *img: tcod.image.Image*, *x: int*, *y: int*) → None

Dynamically update the current font 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.

Deprecated since version 11.13: This function is not supported by contexts. Use `Tileset.set_tile` instead to update tiles.

`tcod.sys_register_SDL_renderer`(*callback: Callable[[Any], None]*) → None
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.

Deprecated since version 11.13: This function is not supported by contexts.

`tcod.sys_check_for_event`(*mask: int, k: Optional[tcod.libtcodpy.Key], m: Optional[tcod.libtcodpy.Mouse]*) → int
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.

Deprecated since version 9.3: Use the `tcod.event.get` function to check for events.

`tcod.sys_wait_for_event`(*mask: int, k: Optional[tcod.libtcodpy.Key], m: Optional[tcod.libtcodpy.Mouse], flush: bool*) → int
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.

Deprecated since version 9.3: Use the `tcod.event.wait` function to wait for events.

17.6 pathfinding

`tcod.dijkstra_compute`(*p: tcod.path.Dijkstra, ox: int, oy: int*) → None

`tcod.dijkstra_delete`(*p: tcod.path.Dijkstra*) → None
Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

`tcod.dijkstra_get`(*p: tcod.path.Dijkstra*, *idx: int*) → Tuple[int, int]

`tcod.dijkstra_get_distance`(*p: tcod.path.Dijkstra*, *x: int*, *y: int*) → int

`tcod.dijkstra_is_empty`(*p: tcod.path.Dijkstra*) → bool

`tcod.dijkstra_new`(*m: tcod.map.Map*, *dcost: float = 1.41*) → tcod.path.Dijkstra

`tcod.dijkstra_new_using_function`(*w: int, h: int, func: Callable[[int, int, int, int, Any], float], userData: Any = 0, dcost: float = 1.41*) → tcod.path.Dijkstra

`tcod.dijkstra_path_set`(*p: tcod.path.Dijkstra*, *x: int*, *y: int*) → bool

`tcod.dijkstra_path_walk`(*p: tcod.path.Dijkstra*) → Union[Tuple[int, int], Tuple[None, None]]

`tcod.dijkstra_reverse`(*p: tcod.path.Dijkstra*) → None

`tcod.dijkstra_size`(*p: tcod.path.Dijkstra*) → int

`tcod.path_compute`(*p: tcod.path.AStar*, *ox: int*, *oy: int*, *dx: int*, *dy: int*) → bool

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: tcod.path.AStar*) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

`tcod.path_get`(*p: tcod.path.AStar*, *idx: int*) → Tuple[int, int]

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: tcod.path.AStar*) → Tuple[int, int]

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: tcod.path.AStar*) → Tuple[int, int]

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: tcod.path.AStar*) → bool

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: int, h: int, func: Callable[[int, int, int, int, Any], float], userData: Any = 0, dcost: float = 1.41*) → tcod.path.AStar

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: tcod.map.Map, dcost: float = 1.41*) → tcod.path.AStar

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: tcod.path.AStar*) → None

Reverse the direction of a path.

This effectively swaps the origin and destination points.

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

`tcod.path_size`(*p: tcod.path.AStar*) → int

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: tcod.path.AStar, recompute: bool*) → Union[Tuple[int, int], Tuple[None, None]]

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]]

17.7 heightmap

tcod.**heightmap_add**(*hm: numpy.ndarray*, *value: float*) → None

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: numpy.ndarray*, *noise: tcod.noise.Noise*, *mulx: float*, *muly: float*, *addx: float*, *addy: float*, *octaves: float*, *delta: float*, *scale: float*) → None

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.

Deprecated since version 8.1: An equivalent array of noise samples can be taken using a method such as Noise.sample_ogrid.

tcod.**heightmap_add_hill**(*hm: numpy.ndarray*, *x: float*, *y: float*, *radius: float*, *height: float*) → None

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: numpy.ndarray*, *hm2: numpy.ndarray*, *hm3: numpy.ndarray*) → None

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: numpy.ndarray, nbPoints: Any, nbCoef: int, coef: Sequence[float], rnd: Optional[tcod.random.Random] = None*) → 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: numpy.ndarray*, *mi: float*, *ma: float*) → None

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: numpy.ndarray*) → None

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: numpy.ndarray*, *hm2: numpy.ndarray*) → None

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: numpy.ndarray*, *mi: float*, *ma: float*) → int

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

Deprecated since version 8.1: Can be replaced by an equivalent NumPy function such as: `numpy.count_nonzero((mi <= hm) & (hm < ma))`

`tcod.heightmap_delete`(*hm: Any*) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

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

`tcod.heightmap_dig_bezier`(*hm: numpy.ndarray, px: Tuple[int, int, int, int], py: Tuple[int, int, int, int], startRadius: float, startDepth: float, endRadius: float, endDepth: float*) → None

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: numpy.ndarray, x: float, y: float, radius: float, height: float*) → None

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: numpy.ndarray, x: float, y: float*) → float

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: numpy.ndarray*) → Tuple[float, float]

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: Use `hm.min()` or `hm.max()` instead.

`tcod.heightmap_get_normal`(*hm: numpy.ndarray, x: float, y: float, waterLevel: float*) → Tuple[float, float, float]

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: numpy.ndarray, x: int, y: int*) → float

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: numpy.ndarray, x: int, y: int*) → float

Return the value at `x`, `y` in a heightmap.

Deprecated since version 2.0: Access *hm* as a NumPy array instead.

`tcod.heightmap_has_land_on_border`(*hm: numpy.ndarray, waterlevel: float*) → bool

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: numpy.ndarray, kernelsize: int, dx: Sequence[int], dy: Sequence[int], weight: Sequence[float], minLevel: float, maxLevel: float*) → None

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: numpy.ndarray, hm2: numpy.ndarray, hm3: numpy.ndarray, coef: float*) → None

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: numpy.ndarray, hm2: numpy.ndarray, hm3: numpy.ndarray*) → None

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: int*, *h: int*, *order: str = 'C'*) → numpy.ndarray

Return a new numpy.ndarray formatted for use with heightmap functions.

w and *h* are the width and height of the array.

order is given to the new NumPy array, it can be 'C' or 'F'.

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

The returned NumPy array will fit all these conditions.

Changed in version 8.1: Added the *order* parameter.

`tcod.`**`heightmap_normalize`**(*hm: numpy.ndarray*, *mi: float = 0.0*, *ma: float = 1.0*) → None

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: numpy.ndarray*, *nbDrops: int*, *erosionCoef: float*, *sedimentationCoef: float*, *rnd: Optional[tcod.random.Random] = None*) → 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: numpy.ndarray*, *value: float*) → None

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: numpy.ndarray*, *noise: tcod.noise.Noise*, *mulx: float*, *muly: float*, *addx: float*, *addy: float*, *octaves: float*, *delta: float*, *scale: float*) → None

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.

Deprecated since version 8.1: An equivalent array of noise samples can be taken using a method such as *Noise.sample_ogrid*.

tcod.**heightmap_set_value**(*hm: numpy.ndarray*, *x: int*, *y: int*, *value: float*) → None
Set the value of a point on a heightmap.

Deprecated since version 2.0: *hm* is a NumPy array, so values should be assigned to it directly.

17.8 image

tcod.**image_load**(*filename: str*) → tcod.image.Image
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: tcod.console.Console*) → tcod.image.Image
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: tcod.image.Image*, *console: tcod.console.Console*, *x: float*, *y: float*, *bkgnd_flag: int*, *scalex: float*, *scaley: float*, *angle: float*) → None

tcod.**image_blit_2x**(*image: tcod.image.Image*, *console: tcod.console.Console*, *dx: int*, *dy: int*, *sx: int = 0*, *sy: int = 0*, *w: int = -1*, *h: int = -1*) → None

tcod.**image_blit_rect**(*image: tcod.image.Image*, *console: tcod.console.Console*, *x: int*, *y: int*, *w: int*, *h: int*, *bkgnd_flag: int*) → None

tcod.**image_clear**(*image: tcod.image.Image, col: Tuple[int, int, int]*) → None

tcod.**image_delete**(*image: tcod.image.Image*) → None
Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

tcod.**image_get_alpha**(*image: tcod.image.Image*, *x: int*, *y: int*) → int

tcod.**image_get_mipmap_pixel**(*image: tcod.image.Image*, *x0: float*, *y0: float*, *x1: float*, *y1: float*) → Tuple[int, int, int]

tcod.**image_get_pixel**(*image: tcod.image.Image*, *x: int*, *y: int*) → Tuple[int, int, int]

tcod.**image_get_size**(*image: tcod.image.Image*) → Tuple[int, int]

tcod.**image_hflip**(*image: tcod.image.Image*) → None

`tcod.image_invert`(*image: tcod.image.Image*) → None

`tcod.image_is_pixel_transparent`(*image: tcod.image.Image*, *x: int*, *y: int*) → bool

`tcod.image_new`(*width: int*, *height: int*) → tcod.image.Image

`tcod.image_put_pixel`(*image: tcod.image.Image, x: int, y: int, col: Tuple[int, int, int]*) → None

`tcod.image_refresh_console`(*image: tcod.image.Image*, *console: tcod.console.Console*) → None

`tcod.image_rotate90`(*image: tcod.image.Image*, *num: int = 1*) → None

`tcod.image_save`(*image: tcod.image.Image*, *filename: str*) → None

`tcod.image_scale`(*image: tcod.image.Image*, *neww: int*, *newh: int*) → None

`tcod.image_set_key_color`(*image: tcod.image.Image, col: Tuple[int, int, int]*) → None

`tcod.image_vflip`(*image: tcod.image.Image*) → None

17.9 line

`tcod.line_init`(*xo: int*, *yo: int*, *xd: int*, *yd: int*) → None

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`() → Union[Tuple[int, int], Tuple[None, None]]

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: int, yo: int, xd: int, yd: int, py_callback: Callable[[int, int], bool]*) → bool

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: int*, *yo: int*, *xd: int*, *yd: int*) → Iterator[Tuple[int, int]]

returns an Iterable

This Iterable 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 Iterable of (x,y) points.

Return type Iterable[Tuple[int,int]]

Deprecated since version 11.14: This function was replaced by `tcod.los.bresenham`.

`tcod.line_where`(*x1: int*, *y1: int*, *x2: int*, *y2: int*, *inclusive: bool = True*) → Tuple[numpy.ndarray, numpy.ndarray]

Return a NumPy index array following a Bresenham line.

If *inclusive* is true then the start point is included in the result.

Example

```
>>> where = tcod.line_where(1, 0, 3, 4)
>>> where
(array([1, 1, 2, 2, 3]...), array([0, 1, 2, 3, 4]...))
>>> array = np.zeros((5, 5), dtype=np.int32)
>>> array[where] = np.arange(len(where[0])) + 1
>>> array
array([[0, 0, 0, 0, 0],
       [1, 2, 0, 0, 0],
       [0, 0, 3, 4, 0],
       [0, 0, 0, 0, 5],
       [0, 0, 0, 0, 0]]...)
```

New in version 4.6.

Deprecated since version 11.14: This function was replaced by `tcod.los.bresenham`.

17.10 map

`tcod.map_clear`(*m: tcod.map.Map*, *transparent: bool = False*, *walkable: bool = False*) → None

Change all map cells to a specific value.

Deprecated since version 4.5: Use `tcod.map.Map.transparent` and `tcod.map.Map.walkable` arrays to set these properties.

`tcod.map_compute_fov`(*m: tcod.map.Map*, *x: int*, *y: int*, *radius: int = 0*, *light_walls: bool = True*, *algo: int = 12*) → None

Compute the field-of-view for a map instance.

Deprecated since version 4.5: Use `tcod.map.Map.compute_fov` instead.

`tcod.map_copy`(*source: tcod.map.Map*, *dest: tcod.map.Map*) → None

Copy map data from *source* to *dest*.

Deprecated since version 4.5: Use Python's copy module, or see `tcod.map.Map` and assign between array attributes manually.

`tcod.map_delete`(*m: tcod.map.Map*) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

`tcod.map_get_height`(*map: tcod.map.Map*) → int

Return the height of a map.

Deprecated since version 4.5: Check the `tcod.map.Map.height` attribute instead.

`tcod.map_get_width`(*map: tcod.map.Map*) → int

Return the width of a map.

Deprecated since version 4.5: Check the `tcod.map.Map.width` attribute instead.

`tcod.map_is_in_fov`(*m: tcod.map.Map*, *x: int*, *y: int*) → bool

Return True if the cell at x,y is lit by the last field-of-view algorithm.

Note: This function is slow.

Deprecated since version 4.5: Use `tcod.map.Map.fov` to check this property.

`tcod.map_is_transparent`(*m: tcod.map.Map*, *x: int*, *y: int*) → bool

Note: This function is slow.

Deprecated since version 4.5: Use `tcod.map.Map.transparent` to check this property.

`tcod.map_is_walkable`(*m: tcod.map.Map*, *x: int*, *y: int*) → bool

Note: This function is slow.

Deprecated since version 4.5: Use `tcod.map.Map.walkable` to check this property.

`tcod.map_new`(*w: int*, *h: int*) → tcod.map.Map
Return a `tcod.map.Map` with a width and height.

Deprecated since version 4.5: Use the `tcod.map` module for working with field-of-view, or `tcod.path` for working with path-finding.

`tcod.map_set_properties`(*m: tcod.map.Map*, *x: int*, *y: int*, *isTrans: bool*, *isWalk: bool*) → None
Set the properties of a single cell.

Note: This function is slow.

Deprecated since version 4.5: Use `tcod.map.Map.transparent` and `tcod.map.Map.walkable` arrays to set these properties.

17.11 mouse

`tcod.mouse_get_status`() → tcod.libtcodpy.Mouse

`tcod.mouse_is_cursor_visible`() → bool
Return True if the mouse cursor is visible.

`tcod.mouse_move`(*x: int*, *y: int*) → None

`tcod.mouse_show_cursor`(*visible: bool*) → None
Change the visibility of the mouse cursor.

17.12 namegen

`tcod.namegen_destroy`() → None

`tcod.namegen_generate`(*name: str*) → str

`tcod.namegen_generate_custom`(*name: str*, *rule: str*) → str

`tcod.namegen_get_sets`() → List[str]

`tcod.namegen_parse`(*filename: str*, *random: Optional[tcod.random.Random] = None*) → None

17.13 noise

`tcod.noise_delete`(*n: tcod.noise.Noise*) → None
Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

`tcod.noise_get`(*n: tcod.noise.Noise, f: Sequence[float], typ: int = 0*) → float
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: tcod.noise.Noise, f: Sequence[float], oc: float, typ: int = 0*) → float

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: tcod.noise.Noise, f: Sequence[float], oc: float, typ: int = 0*) → float

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: int, h: float = 0.5, l: float = 2.0, random: Optional[tcod.random.Random] = None*) → tcod.noise.Noise

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: tcod.noise.Noise, typ: int*) → None

Set a Noise objects default noise algorithm.

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

17.14 parser

`tcod.parser_delete`(*parser: Any*) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

`tcod.parser_get_bool_property`(*parser: Any*, *name: str*) → bool

`tcod.parser_get_char_property`(*parser: Any*, *name: str*) → str

`tcod.parser_get_color_property`(*parser: Any*, *name: str*) → tcod.color.Color

`tcod.parser_get_dice_property`(*parser: Any*, *name: str*) → tcod.libtcodpy.Dice

`tcod.parser_get_float_property`(*parser: Any*, *name: str*) → float

`tcod.parser_get_int_property`(*parser: Any*, *name: str*) → int

`tcod.parser_get_list_property`(*parser: Any*, *name: str*, *type: Any*) → Any

`tcod.parser_get_string_property`(*parser: Any*, *name: str*) → str

`tcod.parser_new`() → Any

`tcod.parser_new_struct`(*parser: Any*, *name: str*) → Any

`tcod.parser_run`(*parser: Any*, *filename: str*, *listener: Any = None*) → None

17.15 random

`tcod.random_delete`(*rnd: tcod.random.Random*) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

`tcod.random_get_double`(*rnd: Optional[tcod.random.Random], mi: float, ma: float*) → float

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: Optional[tcod.random.Random], mi: float, ma: float, mean: float*) → float

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: Optional[tcod.random.Random], mi: float, ma: float*) → float

Return a random float in the range: `mi` <= n <= `ma`.

The result is affected 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: Optional[tcod.random.Random], mi: float, ma: float, mean: float*) → float

Return a random weighted float in the range: `mi` <= n <= `ma`.

The result is affected 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`() → tcod.random.Random

Return the default Random instance.

Returns A Random instance using the default random number generator.

Return type *Random*

`tcod.random_get_int`(*rnd: Optional[tcod.random.Random], mi: int, ma: int*) → int

Return a random integer in the range: `mi` <= n <= `ma`.

The result is affected 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: Optional[tcod.random.Random], mi: int, ma: int, mean: int*) → int

Return a random weighted integer in the range: `mi` <= n <= `ma`.

The result is affected 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: int = 1*) → tcod.random.Random
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: Hashable, algo: int = 1*) → tcod.random.Random
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: Optional[tcod.random.Random], backup: tcod.random.Random*) → None
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.

Deprecated since version 8.4: You can use the standard library copy and pickle modules to save a random state.

`tcod.random_save`(*rnd: Optional[tcod.random.Random]*) → tcod.random.Random
Return a copy of a random number generator.

Deprecated since version 8.4: You can use the standard library copy and pickle modules to save a random state.

`tcod.random_set_distribution`(*rnd: Optional[tcod.random.Random], dist: int*) → None
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_*.

17.16 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*)

17.17 other

class tcod.**ConsoleBuffer**(*width: int*, *height: int*, *back_r: int = 0*, *back_g: int = 0*, *back_b: int = 0*, *fore_r: int = 0*, *fore_g: int = 0*, *fore_b: int = 0*, *char: str = ' '*)

Simple console that allows direct (fast) access to cells. simplifies use of the "fill" functions.

Deprecated since version 6.0: Console array attributes perform better than this class.

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: tcod.console.Console*, *fill_fore: bool = True*, *fill_back: bool = True*) → None

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: int = 0*, *back_g: int = 0*, *back_b: int = 0*, *fore_r: int = 0*, *fore_g: int = 0*, *fore_b: int = 0*, *char: str = ' '*) → None

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() → tcod.libtcodpy.ConsoleBuffer

Returns a copy of this ConsoleBuffer.

Returns A new ConsoleBuffer copy.

Return type *ConsoleBuffer*

set(*x: int*, *y: int*, *back_r: int*, *back_g: int*, *back_b: int*, *fore_r: int*, *fore_g: int*, *fore_b: int*, *char: str*) → None

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: int*, *y: int*, *r: int*, *g: int*, *b: int*) → None

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.

set_fore(*x: int*, *y: int*, *r: int*, *g: int*, *b: int*, *char: str*) → None

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 18

tdl

Deprecated since version 8.4: The *python-tdl* module has been a total disaster and now exists mainly as a historical curiosity and as a stepping stone for what would eventually become *python-tcod*.

18.1 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.

18.2 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`.

18.3 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.

18.4 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='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.

18.5 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

__contains__(*position*)

Use `((x, y) in console)` to check if a position is drawable on this console.

__del__()

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

__iter__()

Return an iterator with every possible (x, y) value for this console.

It goes without saying that working on the console this way is a slow process, especially for Python, and should be minimized.

Returns An ((x, y), . . .) iterator.

Return type Iterator[Tuple[int, int]]

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*

18.6 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.

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*

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

CHAPTER 19

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.`**`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.'

class tdl.event.**Quit**

Fired when the window is closed by the user.

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

True if alt was held down during this event.

Type bool

char = None

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.

Type Text

control = None

True if control was held down during this event.

Type bool

key = None

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*.

Type Text

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.

left_alt = None

type: bool

left_control = None

type: bool

right_alt = None

type: bool

right_control = None

type: bool

shift = None

True if shift was held down during this event.

Type bool

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.**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.**MouseButtonEvent**(*button*, *pos*, *cell*)

Base class for mouse button events.

button = None

Can be one of 'LEFT', 'MIDDLE', 'RIGHT', 'SCROLLUP', 'SCROLLDOWN'

Type Text

cell = None

(x, y) position of the mouse snapped to a cell on the root console

Type Tuple[int, int]

pos = None

(x, y) position of the mouse on the screen.

Type Tuple[int, int]

class tdl.event.**MouseDown**(*button*, *pos*, *cell*)

Fired when a mouse button is pressed.

class tdl.event.**MouseUp**(*button*, *pos*, *cell*)

Fired when a mouse button is released.

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.**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.

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.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.

`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.is_window_closed()`

Returns True if the exit button on the window has been clicked and stays True afterwards.

Returns: bool:

CHAPTER 20

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 creating 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*, *order='F'*)

Field-of-view and path-finding on stored data.

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

Changed in version 4.3: Added *order* parameter.

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 21

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 22

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

V

W

X

Y