python-tcod Documentation

Release 16.2.2

Kyle Benesch

Feb 24, 2024

CONTENTS

Contents:

CHAPTER

ONE

INSTALLATION

Once python-tcod is installed, you'll be able to import the *tcod* module.

The latest version of Python 3 is recommended for a normal install. Python 2 can not be used with the latest versions of python-tcod. These instructions include installing Python if you don't have it yet.

1.1 Windows

First install the latest recent version of Python 3.

Important: Make sure `Add Python to environment variables` is checked in the installer. Otherwise Python will not be added to the Windows `PATH`. If you forgot to do this then you can reopen the installer and *modify* your installation.

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.

You should verify your Python install with your terminal. The terminal you use can be the Windows Command Prompt, PowerShell, GitBash, or similar. It can not be the Python interpreter (indicated with a >>> prompt.) Run the following commands (excluding the >) to verify your Python installation:

```
>python -V
Python 3.10.0

>pip -V
pip 21.2.4 from ...\Python310\lib\site-packages\pip (python 3.10)
```

The above outputs would be the result of Python 3.10 being installed. **Make sure the mentioned Python versions you get are not different than the latest version you just installed.**

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

```
>pip install tcod
```

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

You can then verify that `tcod` is importable from the Python interpreter:

```
>python

>>> import tcod.context
```

If `import tcod.context` doesn't throw an `ImportError` then `tcod` has been installed correctly to your system libraries.

Some IDE's such as PyCharm will create a virtual environment which will ignore your system libraries and require tcod to be installed again in that new environment.

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
```

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 dependencies 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.

From this terminal you can install `tcod` to the virtual environment with the following command:

```
pip install tcod
```

You can now use `import tcod`.

If you are working with multiple people or computers or are using a Git repository then it is 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

TWO

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*.

tcod
tcod on its own is shorthand for both *libtcod* and all of its bindings including *python-tcod*.

It originated as an acronym for the game the library was first created for: The Chronicles Of Doryen

libtcod
This is the original C library which contains the implementations and algorithms used by C programs.

python-tcod includes a statically linked version of this library.

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 the main Python port of *libtcod*.

Originally a superset of the *libtcodpy* API. The major additions included 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.

color control
color controls
Libtcod's old system which assigns colors to specific codepoints. See *libtcodpy.COLCTRL_STOP*, *libtcodpy.COLCTRL_FORE_RGB*, and *libtcodpy.COLCTRL_BACK_RGB* for examples.

CHAPTER

THREE

CHANGELOG

You can find the most recent changelog here.

CHAPTER

FOUR

FREQUENTLY ASKED QUESTIONS

4.1 How do you set a frames-per-second while using contexts?

You'll need to use an external tool to manage the framerate. This can either be your own custom tool or you can copy the Clock class from the framerate.py example.

4.2 I get `No module named 'tcod'` when I try to `import tcod` in PyCharm.

PyCharm will automatically setup a Python virtual environment for new or added projects. By default this virtual environment is isolated and will ignore global Python packages installed from the standard terminal. **In this case you MUST install tcod inside of your per-project virtual environment.**

The recommended way to work with PyCharm is to add a `requirements.txt` file to the root of your PyCharm project with a requirement specifier for *tcod*. This file should have the following:

```
# requirements.txt
# https://pip.pypa.io/en/stable/cli/pip_install/#requirements-file-format
tcod
```

Once this file is saved to your projects root directory then PyCharm will detect it and ask if you want these requirements installed. Say yes and *tcod* will be installed to the *virtual environment*. Be sure to add more specifiers for any modules you're using other than *tcod*, such as *numpy*.

Alternatively you can open the *Terminal* tab in PyCharm and run `pip install tcod` there. This will install *tcod* to the currently open project.

4.3 How do I add custom tiles?

Libtcod uses Unicode to identify tiles. To prevent conflicts with real glyphs you should decide on codepoints from a Private Use Area before continuing. If you're unsure, then use the codepoints from `0x100000` to `0x10FFFD` for your custom tiles.

Normally you load a font with *`tcod.tileset.load_tilesheet()`* which will return a *`Tileset`* that gets passed to *`tcod.context.new()`*'s *tileset* parameter. *`tcod.tileset.load_tilesheet()`* assigns the codepoints from *charmap* to the tilesheet in row-major order.

There are two ways to extend a tileset like the above:

- Increase the tilesheet size vertically and update the *rows* parameter in *tcod.tileset.load_tilesheet()* to match the new image size, then modify the *charmap* parameter to map the new tiles to codepoints. If you edited a CP437 tileset this way then you'd add your new codepoints to the end of *tcod.tileset.CHARMAP_CP437* before using the result as the *charmap* parameter. You can also use *Tileset.remap* if you want to reassign tiles based on their position rather than editing *charmap*.
- Or do not modify the original tilesheet. Load the tileset normally, then add new tiles with *Tileset.set_tile* with manually loaded images.

CHAPTER

FIVE

TUTORIAL

Note: This tutorial is still a work-in-progress. The resources being used are tracked here. Feel free to discuss this tutorial or share your progress on the Github Discussions forum.

Note: This a Python tutorial reliant on a Modern ECS implementation. In this case tcod-ecs will be used. Most other Python ECS libraries do not support entity relationships and arbitrary tags required by this tutorial. If you wish to use this tutorial with another language you may need a Modern ECS implementation on par with Flecs.

5.1 Part 0 - Setting up a project

Note: This tutorial is still a work-in-progress. The resources being used are tracked here. Feel free to discuss this tutorial or share your progress on the Github Discussions forum.

5.1.1 Starting tools

The IDE used for this tutorial is Visual Studio Code[1] (not to be mistaken for Visual Studio).

Git will be used for version control. Follow the instructions here.

Python 3.11 was used to make this tutorial. Get the latest version of Python here. If there exists a version of Python later then 3.11 then install that version instead.

5.1.2 First script

First start with a modern top-level script. Create a script in the project root folder called `main.py` which checks `if __name__ == "__main__":` and calls a `main` function. Any modern script using type-hinting will also have `from __future__ import annotations` near the top.

```
from __future__ import annotations
```

(continues on next page)

[1] Alternatives like PyCharm were considered, but VSCode works the best with Git projects since workspace settings are portable and can be committed without issues.

(continued from previous page)

```
def main() -> None:
    print("Hello World!")


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

In VSCode on the left sidebar is a **Run and Debug** tab. On this tab select **create a launch.json** file. This will prompt about what kind of program to launch. Pick `Python`, then `Module`, then when asked for the module name type `main`. From now on the `F5` key will launch `main.py` in debug mode.

Run the script now and `Hello World!` should be visible in the terminal output.

5.2 Part 1 - Moving a player around the screen

Note: This tutorial is still a work-in-progress. The resources being used are tracked here. Feel free to discuss this tutorial or share your progress on the Github Discussions forum.

In part 1 you will become familiar with the initialization, rendering, and event system of tcod. This will be done as a series of small implementations. It is recommend to save your progress after each section is finished and tested.

5.2.1 Initial script

First start with a modern top-level script. You should have `main.py` script from *Part 0 - Setting up a project*:

```
from __future__ import annotations


def main() -> None:
    ...


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

You will replace body of the `main` function in the following section.

5.2.2 Loading a tileset and opening a window

From here it is time to setup a `tcod` program. Download Alloy_curses_12x12.png[1] and place this file in your projects `data/` directory. This tileset is from the Dwarf Fortress tileset repository. These kinds of tilesets are always loaded with `columns=16, rows=16, charmap=tcod.tileset.CHARMAP_CP437`. Use the string `"data/Alloy_curses_12x12.png"` to refer to the path of the tileset.[2]

[1] The choice of tileset came down to what looked nice while also being square. Other options such as using a BDF font were considered, but in the end this tutorial won't go too much into Unicode.

[2] `pathlib` is not used because this example is too simple for that. The working directory will always be the project root folder for the entire tutorial, including distributions. `pathlib` will be used later for saved games and configuration directories, and not for static data.

Load the tileset with *tcod.tileset.load_tilesheet*. Pass the tileset to *tcod.tileset.procedural_block_elements* which will fill in most Block Elements missing from Code Page 437. Then pass the tileset to *tcod.context.new*, you only need to provide the `tileset` parameter.

tcod.context.new returns a *Context* which will be used with Python's **with** statement. We want to keep the name of the context, so use the syntax: **with** `tcod.context.new(tileset=tileset)` **as** `context:`. The new block can't be empty, so add **pass** to the with statement body.

These functions are part of modules which have not been imported yet, so new imports for `tcod.context` and `tcod.tileset` must be added to the top of the script.

```
from __future__ import annotations

import tcod.context  # Add these imports
import tcod.tileset


def main() -> None:
    """Load a tileset and open a window using it, this window will immediately close."""
    tileset = tcod.tileset.load_tilesheet(
        "data/Alloy_curses_12x12.png", columns=16, rows=16, charmap=tcod.tileset.CHARMAP_
↪CP437
    )
    tcod.tileset.procedural_block_elements(tileset=tileset)
    with tcod.context.new(tileset=tileset) as context:
        pass  # The window will stay open for the duration of this block


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

If an import fails that means you do not have `tcod` installed on the Python environment you just used to run the script. If you use an IDE then make sure the Python environment it is using is correct and then run `pip install tcod` from the shell terminal within that IDE.

There is no game loop, so if you run this script now then a window will open and then immediately close. If that happens without seeing a traceback in your terminal then the script is correct.

5.2.3 Configuring an event loop

The next step is to keep the window open until the user closes it.

Since nothing is displayed yet a *Console* should be created with `"Hello World"` printed to it. The size of the console can be used as a reference to create the context by adding the console to *tcod.context.new*.[3]

Begin the main game loop with a **while** `True:` statement.

To actually display the console to the window the *Context.present* method must be called with the console as a parameter. Do this first in the game loop before handing events.

Events are checked by iterating over all pending events with *tcod.event.wait*. Use the code **for** `event` **in** `tcod.event.wait():` to begin handing events.

In the event loop start with the line `print(event)` so that all events can be viewed from the program output. Then test if an event is for closing the window with **if** `isinstance(event, tcod.event.Quit):`. If this is True then

[3] This tutorial follows the setup for a fixed-size console. The alternatives shown in *Getting Started* are outside the scope of this tutorial.

you should exit the function with `raise SystemExit()`.[4]

```
from __future__ import annotations

import tcod.console
import tcod.context
import tcod.event
import tcod.tileset


def main() -> None:
    """Show "Hello World" until the window is closed."""
    tileset = tcod.tileset.load_tilesheet(
        "data/Alloy_curses_12x12.png", columns=16, rows=16, charmap=tcod.tileset.CHARMAP_
↪CP437
    )
    tcod.tileset.procedural_block_elements(tileset=tileset)
    console = tcod.console.Console(80, 50)
    console.print(0, 0, "Hello World")  # Test text by printing "Hello World" to the␣
↪console
    with tcod.context.new(console=console, tileset=tileset) as context:
        while True:  # Main loop
            context.present(console)  # Render the console to the window and show it
            for event in tcod.event.wait():  # Event loop, blocks until pending events␣
↪exist
                print(event)
                if isinstance(event, tcod.event.Quit):
                    raise SystemExit()


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

If you run this then you get a window saying `"Hello World"`. The window can be resized and the console will be stretched to fit the new resolution. When you do anything such as press a key or interact with the window the event for that action will be printed to the program output.

5.2.4 An example game state

What exists now is not very interactive. The next step is to change state based on user input.

Like `tcod` you'll need to install `attrs` with Pip, such as with `pip install attrs`.

Start by adding an `attrs` class called `ExampleState`. This a normal class with the `@attrs.define(eq=False)` decorator added.

This class should hold coordinates for the player. It should also have a `on_draw` method which takes *`tcod.console.Console`* as a parameter and marks the player position on it. The parameters for `on_draw` are `self` because this is an instance method and `console: tcod.console.Console`. `on_draw` returns nothing, so be sure to add `-> None`.

`Console.print` is the simplest way to draw the player because other options would require bounds-checking. Call this method using the players current coordinates and the `"@"` character.

[4] You could use `return` here to exit the `main` function and end the program, but `raise SystemExit()` is used because it will close the program from anywhere. `raise SystemExit()` is also more useful to teach than `sys.exit`.

```
from __future__ import annotations

import attrs
import tcod.console
import tcod.context
import tcod.event
import tcod.tileset


@attrs.define(eq=False)
class ExampleState:
    """Example state with a hard-coded player position."""

    player_x: int
    """Player X position, left-most position is zero."""
    player_y: int
    """Player Y position, top-most position is zero."""

    def on_draw(self, console: tcod.console.Console) -> None:
        """Draw the player glyph."""
        console.print(self.player_x, self.player_y, "@")

...
```

Now remove the `console.print(0, 0, "Hello World")` line from `main`.

Before the context is made create a new `ExampleState` with player coordinates on the screen. Each *Console* has `.width` and `.height` attributes which you can divide by 2 to get a centered coordinate for the player. Use Python's floor division operator `//` so that the resulting type is `int`.

Modify the drawing routine so that the console is cleared, then passed to `ExampleState.on_draw`, then passed to *Context.present*.

```
...
def main() -> None:
    """Run ExampleState."""
    tileset = tcod.tileset.load_tilesheet(
        "data/Alloy_curses_12x12.png", columns=16, rows=16, charmap=tcod.tileset.CHARMAP_
↪CP437
    )
    tcod.tileset.procedural_block_elements(tileset=tileset)
    console = tcod.console.Console(80, 50)
    state = ExampleState(player_x=console.width // 2, player_y=console.height // 2)
    with tcod.context.new(console=console, tileset=tileset) as context:
        while True:
            console.clear()  # Clear the console before any drawing
            state.on_draw(console)  # Draw the current state
            context.present(console)  # Display the console on the window
            for event in tcod.event.wait():
                print(event)
                if isinstance(event, tcod.event.Quit):
                    raise SystemExit()
```

(continues on next page)

(continued from previous page)

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

Now if you run the script you'll see `@`.

The next step is to move the player on events. A new method will be added to the `ExampleState` for this called `on_event`. `on_event` takes a `self` and a *tcod.event.Event* parameter and returns nothing.

Events are best handled using Python's Structural Pattern Matching. Consider reading Python's Structural Pattern Matching Tutorial.

Begin matching with **match** `event:`. The equivalent to **if** `isinstance(event, tcod.event.Quit):` is **case** `tcod.event.Quit():`. Keyboard keys can be checked with **case** `tcod.event.KeyDown(sym=tcod.event.KeySym.LEFT):`. Make a case for each arrow key: `LEFT RIGHT UP DOWN` and move the player in the direction of that key. Since events are printed you can check the *KeySym* of a key by pressing that key and looking at the printed output. See *KeySym* for a list of all keys.

Finally replace the event handling code in `main` to defer to the states `on_event` method. The full script so far is:

```
from __future__ import annotations

import attrs
import tcod.console
import tcod.context
import tcod.event
import tcod.tileset


@attrs.define(eq=False)
class ExampleState:
    """Example state with a hard-coded player position."""

    player_x: int
    """Player X position, left-most position is zero."""
    player_y: int
    """Player Y position, top-most position is zero."""

    def on_draw(self, console: tcod.console.Console) -> None:
        """Draw the player glyph."""
        console.print(self.player_x, self.player_y, "@")

    def on_event(self, event: tcod.event.Event) -> None:
        """Move the player on events and handle exiting. Movement is hard-coded."""
        match event:
            case tcod.event.Quit():
                raise SystemExit()
            case tcod.event.KeyDown(sym=tcod.event.KeySym.LEFT):
                self.player_x -= 1
            case tcod.event.KeyDown(sym=tcod.event.KeySym.RIGHT):
                self.player_x += 1
            case tcod.event.KeyDown(sym=tcod.event.KeySym.UP):
                self.player_y -= 1
            case tcod.event.KeyDown(sym=tcod.event.KeySym.DOWN):
                self.player_y += 1
```

(continues on next page)

(continued from previous page)

```
def main() -> None:
    """Run ExampleState."""
    tileset = tcod.tileset.load_tilesheet(
        "data/Alloy_curses_12x12.png", columns=16, rows=16, charmap=tcod.tileset.CHARMAP_
↪CP437
    )
    tcod.tileset.procedural_block_elements(tileset=tileset)
    console = tcod.console.Console(80, 50)
    state = ExampleState(player_x=console.width // 2, player_y=console.height // 2)
    with tcod.context.new(console=console, tileset=tileset) as context:
        while True:
            console.clear()
            state.on_draw(console)
            context.present(console)
            for event in tcod.event.wait():
                print(event)
                state.on_event(event)  # Pass events to the state


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

Now when you run this script you have a player character you can move around with the arrow keys before closing the window.

You can review the part-1 source code here.

5.3 Part 2 - Entities

Note: This tutorial is still a work-in-progress. The resources being used are tracked here. Feel free to discuss this tutorial or share your progress on the Github Discussions forum.

In part 2 entities will be added and a new state will be created to handle them. This part will also begin to split logic into multiple Python modules using a namespace called `game`.

Entities will be handled with an ECS implementation, in this case: tcod-ecs. `tcod-ecs` is a standalone package and is installed separately from `tcod`. Use `pip install tcod-ecs` to install this package.

5.3.1 Namespace package

Create a new folder called `game` and inside the folder create a new python file named `__init__.py`. `game/__init__.py` only needs a docstring describing that it is a namespace package:

```
"""Game namespace package."""
```

This package will be used to organize new modules.

5.3.2 Organizing globals

There are a few variables which will need to be accessible from multiple modules. Any global variables which might be assigned from other modules will need to a tracked and handled with care.

Create a new module: `g.py`[1]. This module is exceptional and will be placed at the top-level instead of in the `game` folder.

In `g.py` import `tcod.context` and `tcod.ecs`.

`context` from `main.py` will now be annotated in `g.py` by adding the line `context: tcod.context.Context` by itself. Notice that is this only a type-hinted name and nothing is assigned to it. This means that type-checking will assume the variable always exists but using it before it is assigned will crash at run-time.

`main.py` should add `import g` and replace the variables named `context` with `g.context`.

Then add the `world: tcod.ecs.Registry` global to hold the ECS scope.

It is important to document all variables placed in this module with docstrings.

```
"""This module stores globally mutable variables used by this program."""
from __future__ import annotations

import tcod.context
import tcod.ecs

context: tcod.context.Context
"""The window managed by tcod."""

world: tcod.ecs.Registry
"""The active ECS registry and current session."""
```

Ideally you should not overuse this module for too many things. When a variable can either be taken as a function parameter or accessed as a global then passing as a parameter is always preferable.

5.3.3 ECS tags

Create `game/tags.py`. This will hold some sentinel values to be used as tags for `tcod-ecs`. These tags can be anything that's both unique and unchanging, in this case Python strings are used.

For example `IsPlayer: Final = "IsPlayer"` will tag an object as being controlled by the player. The name is `IsPlayer` and string is the same as the name. The `Final` annotation clarifies that this a constant. Sentinel values for `tcod-ecs` are named like classes, similar to names like `None` or `False`.

Repeat this for `IsActor` and `IsItem` tags. The `game/tags.py` module should look like this:

```
"""Collection of common tags."""
from __future__ import annotations

from typing import Final

IsPlayer: Final = "IsPlayer"
"""Entity is the player."""
```

(continues on next page)

[1] `global`, `globals`, and `glob` were already taken by keywords, built-ins, and the standard library. The alternatives are to either put this in the `game` namespace or to add an underscore such as `globals_.py`.

(continued from previous page)

```
IsActor: Final = "IsActor"
"""Entity is an actor."""

IsItem: Final = "IsItem"
"""Entity is an item."""
```

5.3.4 ECS components

Next is a new `game/components.py` module. This will hold the components for the graphics and position of entities.

Start by adding an import for `attrs`. The ability to easily design small classes which are frozen/immutable is important for working with `tcod-ecs`.

The first component will be a `Position` class. This class will be decorated with `@attrs.define(frozen=True)`. For attributes this class will have `x: int` and `y: int`.

It will be common to add vectors to a `Position` with code such as `new_pos: Position = Position(0, 0) + (0, 1)`. Create the dunder method `def __add__(self, direction: tuple[int, int]) -> Self:` to allow this syntax. Unpack the input with `x, y = direction`. `self.__class__` is the current class so `self.__class__(self.x + x, self.y + y)` will create a new instance with the direction added to the previous values.

The new class will look like this:

```
@attrs.define(frozen=True)
class Position:
    """An entities position."""

    x: int
    y: int

    def __add__(self, direction: tuple[int, int]) -> Self:
        """Add a vector to this position."""
        x, y = direction
        return self.__class__(self.x + x, self.y + y)
```

Because `Position` is immutable, `tcod-ecs` is able to reliably track changes to this component. Normally you can only query entities by which components they have. A callback can be registered with `tcod-ecs` to mirror component values as tags. This allows querying an entity by its exact position.

Add `import tcod.ecs.callbacks` and `from tcod.ecs import Entity`. Then create the new function `def on_position_changed(entity: Entity, old: Position | None, new: Position | None) -> None:` decorated with `@tcod.ecs.callbacks.register_component_changed(component=Position)`. This function is called when the `Position` component is either added, removed, or modified by assignment. The goal of this function is to mirror the current position to the `set`-like attribute `entity.tags`.

`if old == new:` then a position was assigned its own value or an equivalent value. The cost of discarding and adding the same value can sometimes be high so this case should be guarded and ignored. `if old is not None:` then the value tracked by `entity.tags` is outdated and must be removed. `if new is not None:` then `new` is the up-to-date value to be tracked by `entity.tags`.

The function should look like this:

```
@tcod.ecs.callbacks.register_component_changed(component=Position)
def on_position_changed(entity: Entity, old: Position | None, new: Position | None) ->␣
```

(continues on next page)

(continued from previous page)

```
↪None:
    """Mirror position components as a tag."""
    if old == new:  # New position is equivalent to its previous value
        return  # Ignore and return
    if old is not None:  # Position component removed or changed
        entity.tags.discard(old)  # Remove old position from tags
    if new is not None:  # Position component added or changed
        entity.tags.add(new)  # Add new position to tags
```

Next is the `Graphic` component. This will have the attributes `ch: int = ord("!")` and `fg: tuple[int, int, int] = (255, 255, 255)`. By default all new components should be marked as frozen.

```
@attrs.define(frozen=True)
class Graphic:
    """An entities icon and color."""

    ch: int = ord("!")
    fg: tuple[int, int, int] = (255, 255, 255)
```

One last component: `Gold`. Define this as `Gold: Final = ("Gold", int)`. `(name, type)` is tcod-ecs specific syntax to handle multiple components sharing the same type.

```
Gold: Final = ("Gold", int)
"""Amount of gold."""
```

That was the last component. The `game/components.py` module should look like this:

```
"""Collection of common components."""
from __future__ import annotations

from typing import Final, Self

import attrs
import tcod.ecs.callbacks
from tcod.ecs import Entity


@attrs.define(frozen=True)
class Position:
    """An entities position."""

    x: int
    y: int

    def __add__(self, direction: tuple[int, int]) -> Self:
        """Add a vector to this position."""
        x, y = direction
        return self.__class__(self.x + x, self.y + y)


@tcod.ecs.callbacks.register_component_changed(component=Position)
def on_position_changed(entity: Entity, old: Position | None, new: Position | None) ->␣
↪None:
```

(continues on next page)

(continued from previous page)

```
    """Mirror position components as a tag."""
    if old == new:
        return
    if old is not None:
        entity.tags.discard(old)
    if new is not None:
        entity.tags.add(new)


@attrs.define(frozen=True)
class Graphic:
    """An entities icon and color."""

    ch: int = ord("!")
    fg: tuple[int, int, int] = (255, 255, 255)


Gold: Final = ("Gold", int)
"""Amount of gold."""
```

5.3.5 ECS entities and registry

Now it is time to create entities. To do that you need to create the ECS registry.

Make a new script called `game/world_tools.py`. This module will be used to create the ECS registry.

Random numbers from `random` will be used. In this case we want to use `Random` as a component so add `from random import Random`. Get the registry with `from tcod.ecs import Registry`. Collect all our components and tags with `from game.components import Gold, Graphic, Position` and `from game.tags import IsActor, IsItem, IsPlayer`.

This module will have one function: `def new_world() -> Registry:`. Think of the ECS registry as containing the world since this is how it will be used. Start this function with `world = Registry()`.

Entities are referenced with the syntax `world[unique_id]`. If the same `unique_id` is used then you will access the same entity. `new_entity = world[object()]` is the syntax to spawn new entities because `object()` is always unique. Whenever a global entity is needed then `world[None]` will be used.

Create an instance of `Random()` and assign it to both `world[None].components[Random]` and `rng`. This can done on one line with `rng = world[None].components[Random] = Random()`.

Next create the player entity with `player = world[object()]`. Assign the following components to the new player entity: `player.components[Position] = Position(5, 5)`, `player.components[Graphic] = Graphic(ord("@"))`, and `player.components[Gold] = 0`. Then update the players tags with `player.tags |= {IsPlayer, IsActor}`.

To add some variety we will scatter gold randomly across the world. Start a for-loop with `for _ in range(10):` then create a `gold` entity in this loop.

The `Random` instance `rng` has access to functions from Python's random module such as `random.randint`. Set `Position` to `Position(rng.randint(0, 20), rng.randint(0, 20))`. Set `Graphic` to `Graphic(ord("$"), fg=(255, 255, 0))`. Set `Gold` to `rng.randint(1, 10)`. Then add `IsItem` as a tag.

Once the for-loop exits then `return world`. Make sure `return` has the correct indentation and is not part of the for-loop or else you will only spawn one gold.

game/world_tools.py should look like this:

```
"""Functions for working with worlds."""
from __future__ import annotations

from random import Random

from tcod.ecs import Registry

from game.components import Gold, Graphic, Position
from game.tags import IsActor, IsItem, IsPlayer


def new_world() -> Registry:
    """Return a freshly generated world."""
    world = Registry()

    rng = world[None].components[Random] = Random()

    player = world[object()]
    player.components[Position] = Position(5, 5)
    player.components[Graphic] = Graphic(ord("@"))
    player.components[Gold] = 0
    player.tags |= {IsPlayer, IsActor}

    for _ in range(10):
        gold = world[object()]
        gold.components[Position] = Position(rng.randint(0, 20), rng.randint(0, 20))
        gold.components[Graphic] = Graphic(ord("$"), fg=(255, 255, 0))
        gold.components[Gold] = rng.randint(1, 10)
        gold.tags |= {IsItem}

    return world
```

5.3.6 New InGame state

Now there is a new ECS world but the example state does not know how to render it. A new state needs to be made which is aware of the new entities.

Create a new script called game/states.py. states is for derived classes, state is for the abstract class. New states will be created in this module and this module will be allowed to import many first party modules without issues.

Before adding a new state it is time to add a more complete set of directional keys. These will be added as a dictionary and can be reused anytime we want to know how a key translates to a direction. Use from tcod.event import KeySym to make KeySym enums easier to write. Then add the following:

```
DIRECTION_KEYS: Final = {
    # Arrow keys
    KeySym.LEFT: (-1, 0),
    KeySym.RIGHT: (1, 0),
    KeySym.UP: (0, -1),
    KeySym.DOWN: (0, 1),
    # Arrow key diagonals
```

(continues on next page)

(continued from previous page)

```
    KeySym.HOME: (-1, -1),
    KeySym.END: (-1, 1),
    KeySym.PAGEUP: (1, -1),
    KeySym.PAGEDOWN: (1, 1),
    # Keypad
    KeySym.KP_4: (-1, 0),
    KeySym.KP_6: (1, 0),
    KeySym.KP_8: (0, -1),
    KeySym.KP_2: (0, 1),
    KeySym.KP_7: (-1, -1),
    KeySym.KP_1: (-1, 1),
    KeySym.KP_9: (1, -1),
    KeySym.KP_3: (1, 1),
    # VI keys
    KeySym.h: (-1, 0),
    KeySym.l: (1, 0),
    KeySym.k: (0, -1),
    KeySym.j: (0, 1),
    KeySym.y: (-1, -1),
    KeySym.b: (-1, 1),
    KeySym.u: (1, -1),
    KeySym.n: (1, 1),
}
```

Create a new `class InGame:` decorated with `@attrs.define(eq=False)`. States will always use `g.world` to access the ECS registry.

```
@attrs.define(eq=False)
class InGame:
    """Primary in-game state."""
    ...
```

Create an `on_event` and `on_draw` method matching the `ExampleState` class. Copying `ExampleState` and modifying it should be enough since this wil replace `ExampleState`.

Now to do an tcod-ecs query to fetch the player entity. In tcod-ecs queries most often start with `g.world.Q.all_of(components=[], tags=[])`. Which components and tags are asked for will narrow down the returned set of entities to only those matching the requirements. The query to fetch player entities is `g.world.Q.all_of(tags=[IsPlayer])`. We expect only one player so the result will be unpacked into a single name: `(player,) = g.world.Q.all_of(tags=[IsPlayer])`.

Next is to handle the event. Handling `case tcod.event.Quit():` is the same as before: `raise SystemExit()`.

The case for direction keys will now be done in a single case: `case tcod.event.KeyDown(sym=sym) if sym in DIRECTION_KEYS:`. `sym=sym` assigns from the event attribute to a local name. The left side is the `event.sym` attribute and right side is the local name `sym` being assigned to. The case also has a condition which must pass for this branch to be taken and in this case we ensure that only keys from the `DIRECTION_KEYS` dictionary are valid `sym`'s.

Inside this branch moving the player is simple. Access the `(x, y)` vector with `DIRECTION_KEYS[sym]` and use `+=` to add it to the players current `Position` component. This triggers the earlier written `__add__` dunder method and `on_position_changed` callback.

Now that the player has moved it would be a good time to interact with the gold entities. The query to see if the player has stepped on gold is to check for whichever entities have a `Gold` component, an `IsItem` tag, and the players current position as a tag. The query for this is `g.world.Q.all_of(components=[Gold], tags=[player.`

`components[Position], IsItem]):`.

We will iterate over whatever matches this query using a `for gold in ...:` loop. Add the entities `Gold` component to the players similar component. Keep in mind that `Gold` is treated like an `int` so its usage is predictable.

Format the added and total of gold using a Python f-string: `text = f"Picked up {gold.components[Gold]}g, total: {player.components[Gold]}g"`. Store `text` globally in the ECS registry with `g.world[None].components[("Text", str)] = text`. This is done as two lines to avoid creating a line with an excessive length.

Then use `gold.clear()` at the end to remove all components and tags from the gold entity which will effectively delete it.

```
...
def on_event(self, event: tcod.event.Event) -> None:
    """Handle events for the in-game state."""
    (player,) = g.world.Q.all_of(tags=[IsPlayer])
    match event:
        case tcod.event.Quit():
            raise SystemExit()
        case tcod.event.KeyDown(sym=sym) if sym in DIRECTION_KEYS:
            player.components[Position] += DIRECTION_KEYS[sym]
            # Auto pickup gold
            for gold in g.world.Q.all_of(components=[Gold], tags=[player.
↪components[Position], IsItem]):
                player.components[Gold] += gold.components[Gold]
                text = f"Picked up {gold.components[Gold]}g, total: {player.
↪components[Gold]}g"
                g.world[None].components[str] = text
                gold.clear()
...
```

Now start with the `on_draw` method. Any entity with both a `Position` and a `Graphic` is drawable. Iterate over these entities with `for entity in g.world.Q.all_of(components=[Position, Graphic]):`. Accessing components can be slow in a loop, so assign components to local names before using them (`pos = entity.components[Position]` and `graphic = entity.components[Graphic]`).

Check if a components position is in the bounds of the console. `0 <= pos.x < console.width and 0 <= pos.y < console.height` tells if the position is in bounds. Instead of nesting this method further, this check should be a guard using `if not (...):` and `continue`.

Draw the graphic by assigning it to the consoles Numpy array directly with `console.rgb[["ch", "fg"]][pos.y, pos.x] = graphic.ch, graphic.fg`. `console.rgb` is a `ch,fg,bg` array and `[["ch", "fg"]]` narrows it down to only `ch,fg`. The array is in C row-major memory order so you access it with yx (or ij) ordering.

That ends the entity rendering loop. Next is to print the `("Text", str)` component if it exists. A normal access will raise `KeyError` if the component is accessed before being assigned. This case will be handled by the `.get` method of the `Entity.components` attribute. `g.world[None].components.get(("Text", str))` will return `None` instead of raising `KeyError`. Assigning this result to `text` and then checking `if text:` will ensure that `text` within the branch is not None and that the string is not empty. We will not use `text` outside of the branch, so an assignment expression can be used here to check and assign the name at the same time with `if text := g.world[None].components.get(("Text", str)):`.

In this branch you will print `text` to the bottom of the console with a white foreground and black background. The call to do this is `console.print(x=0, y=console.height - 1, string=text, fg=(255, 255, 255), bg=(0, 0, 0))`.

```
...
def on_draw(self, console: tcod.console.Console) -> None:
    """Draw the standard screen."""
    for entity in g.world.Q.all_of(components=[Position, Graphic]):
        pos = entity.components[Position]
        if not (0 <= pos.x < console.width and 0 <= pos.y < console.height):
            continue
        graphic = entity.components[Graphic]
        console.rgb[["ch", "fg"]][pos.y, pos.x] = graphic.ch, graphic.fg

    if text := g.world[None].components.get(("Text", str)):
        console.print(x=0, y=console.height - 1, string=text, fg=(255, 255, 255), bg=(0,␣
↪0, 0))
```

Verify the indentation of the `if` branch is correct. It should be at the same level as the `for` loop and not inside of it.

`game/states.py` should now look like this:

```
"""A collection of game states."""
from __future__ import annotations

from typing import Final

import attrs
import tcod.console
import tcod.event
from tcod.event import KeySym

import g
from game.components import Gold, Graphic, Position
from game.tags import IsItem, IsPlayer

DIRECTION_KEYS: Final = {
    # Arrow keys
    KeySym.LEFT: (-1, 0),
    KeySym.RIGHT: (1, 0),
    KeySym.UP: (0, -1),
    KeySym.DOWN: (0, 1),
    # Arrow key diagonals
    KeySym.HOME: (-1, -1),
    KeySym.END: (-1, 1),
    KeySym.PAGEUP: (1, -1),
    KeySym.PAGEDOWN: (1, 1),
    # Keypad
    KeySym.KP_4: (-1, 0),
    KeySym.KP_6: (1, 0),
    KeySym.KP_8: (0, -1),
    KeySym.KP_2: (0, 1),
    KeySym.KP_7: (-1, -1),
    KeySym.KP_1: (-1, 1),
    KeySym.KP_9: (1, -1),
    KeySym.KP_3: (1, 1),
    # VI keys
```

(continues on next page)

(continued from previous page)

```
    KeySym.h: (-1, 0),
    KeySym.l: (1, 0),
    KeySym.k: (0, -1),
    KeySym.j: (0, 1),
    KeySym.y: (-1, -1),
    KeySym.b: (-1, 1),
    KeySym.u: (1, -1),
    KeySym.n: (1, 1),
}


@attrs.define(eq=False)
class InGame:
    """Primary in-game state."""

    def on_event(self, event: tcod.event.Event) -> None:
        """Handle events for the in-game state."""
        (player,) = g.world.Q.all_of(tags=[IsPlayer])
        match event:
            case tcod.event.Quit():
                raise SystemExit()
            case tcod.event.KeyDown(sym=sym) if sym in DIRECTION_KEYS:
                player.components[Position] += DIRECTION_KEYS[sym]
                # Auto pickup gold
                for gold in g.world.Q.all_of(components=[Gold], tags=[player.
↪components[Position], IsItem]):
                    player.components[Gold] += gold.components[Gold]
                    text = f"Picked up {gold.components[Gold]}g, total: {player.
↪components[Gold]}g"
                    g.world[None].components[("Text", str)] = text
                    gold.clear()

    def on_draw(self, console: tcod.console.Console) -> None:
        """Draw the standard screen."""
        for entity in g.world.Q.all_of(components=[Position, Graphic]):
            pos = entity.components[Position]
            if not (0 <= pos.x < console.width and 0 <= pos.y < console.height):
                continue
            graphic = entity.components[Graphic]
            console.rgb[["ch", "fg"]][pos.y, pos.x] = graphic.ch, graphic.fg

        if text := g.world[None].components.get(("Text", str)):
            console.print(x=0, y=console.height - 1, string=text, fg=(255, 255, 255),␣
↪bg=(0, 0, 0))
```

5.3.7 Main script update

Back to `main.py`. At this point you should know to import the modules needed.

The `ExampleState` class is obsolete and will be removed. `state` will be created with `game.states.InGame()` instead.

If you have not replaced `context` with `g.context` yet then do it now.

Add `g.world = game.world_tools.new_world()` before the main loop.

`main.py` will look like this:

```
#!/usr/bin/env python3
"""Main entry-point module. This script is used to start the program."""
from __future__ import annotations

import tcod.console
import tcod.context
import tcod.event
import tcod.tileset

import g
import game.states
import game.world_tools


def main() -> None:
    """Entry point function."""
    tileset = tcod.tileset.load_tilesheet(
        "data/Alloy_curses_12x12.png", columns=16, rows=16, charmap=tcod.tileset.CHARMAP_
↪CP437
    )
    tcod.tileset.procedural_block_elements(tileset=tileset)
    console = tcod.console.Console(80, 50)
    state = game.states.InGame()
    g.world = game.world_tools.new_world()
    with tcod.context.new(console=console, tileset=tileset) as g.context:
        while True:  # Main loop
            console.clear()  # Clear the console before any drawing
            state.on_draw(console)  # Draw the current state
            g.context.present(console)  # Render the console to the window and show it
            for event in tcod.event.wait():  # Event loop, blocks until pending events␣
↪exist
                print(event)
                state.on_event(event)  # Dispatch events to the state


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

Now you can play a simple game where you wander around collecting gold.

You can review the part-2 source code here.

5.4 Part 3 - UI State

Note: This tutorial is still a work-in-progress. The resources being used are tracked here. Feel free to discuss this tutorial or share your progress on the Github Discussions forum.

Warning: This part is still a draft and is being worked on. Sections here will be incorrect as these examples were hastily moved from an earlier part.

5.4.1 State protocol

To have more states than `ExampleState` one must use an abstract type which can be used to refer to any state. In this case a Protocol will be used, called `State`.

Create a new module: `game/state.py`. In this module add the class `class State(Protocol):`. `Protocol` is from Python's `typing` module. `State` should have the `on_event` and `on_draw` methods from `ExampleState` but these methods will be empty other than the docstrings describing what they are for. These methods refer to types from `tcod` and those types will need to be imported. `State` should also have `__slots__ = ()`[1] in case the class is used for a subclass.

`game/state.py` should look like this:

```
"""Base classes for states."""
from __future__ import annotations

from typing import Protocol

import tcod.console
import tcod.event


class State(Protocol):
    """An abstract game state."""

    __slots__ = ()

    def on_event(self, event: tcod.event.Event) -> None:
        """Called on events."""

    def on_draw(self, console: tcod.console.Console) -> None:
        """Called when the state is being drawn."""
```

The `InGame` class does not need to be updated since it is already a structural subtype of `State`. Note that subclasses of `State` will never be in same module as `State`, this will be the same for all abstract classes.

[1] This is done to prevent subclasses from requiring a `__dict__` attribute. If you are still wondering what `__slots__` is then the Python docs have a detailed explanation.

5.4.2 State globals

A new global will be added: `states: list[game.state.State] = []`. States are implemented as a list/stack to support pushdown automata. Representing states as a stack makes it easier to implement popup windows, menus, and other "history aware" states.

5.4.3 State functions

Create a new module: `game/state_tools.py`. This module will handle events and rendering of the global state.

In this module add the function `def main_draw() -> None:`. This will hold the "clear, draw, present" logic from the `main` function which will be moved to this function. Render the active state with `g.states[-1].on_draw(g.console)`. If `g.states` is empty then this function should immediately `return` instead of doing anything. Empty containers in Python are `False` when checked for truthiness.

Next the function `def main_loop() -> None:` is created. The `while` loop from `main` will be moved to this function. The while loop will be replaced by `while g.states:` so that this function will exit if no state exists. Drawing will be replaced by a call to `main_draw`. Events in the for-loop will be passed to the active state `g.states[-1].on_event(event)`. Any states `on_event` method could potentially change the state so `g.states` must be checked to be non-empty for every handled event.

```
"""State handling functions."""
from __future__ import annotations

import tcod.console

import g


def main_draw() -> None:
    """Render and present the active state."""
    if not g.states:
        return
    g.console.clear()
    g.states[-1].on_draw(g.console)
    g.context.present(g.console)


def main_loop() -> None:
    """Run the active state forever."""
    while g.states:
        main_draw()
        for event in tcod.event.wait():
            if g.states:
                g.states[-1].on_event(event)
```

Now `main.py` can be edited to use the global variables and the new game loop.

Add `import g` and `import game.state_tools`. Replace references to `console` with `g.console`. Replace references to `context` with `g.context`.

States are initialed by assigning a list with the initial state to `g.states`. The previous game loop is replaced by a call to `game.state_tools.main_loop()`.

```
...

import g
import game.state_tools

def main() -> None:
    """Entry point function."""
    tileset = tcod.tileset.load_tilesheet(
        "data/Alloy_curses_12x12.png", columns=16, rows=16, charmap=tcod.tileset.CHARMAP_
↪CP437
    )
    tcod.tileset.procedural_block_elements(tileset=tileset)
    g.console = tcod.console.Console(80, 50)
    g.states = [ExampleState(player_x=console.width // 2, player_y=console.height // 2)]
    with tcod.context.new(console=g.console, tileset=tileset) as g.context:
        game.state_tools.main_loop()
...
```

After this you can test the game. There should be no visible differences from before.

CHAPTER

SIX

GETTING STARTED

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

6.1 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 python
# Make sure 'dejavu10x10_gs_tc.png' is in the same directory as this script.
import tcod.console
import tcod.context
import tcod.event
import tcod.tileset

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

            # This event loop will wait until at least one event is processed before␣
```

(continues on next page)

(continued from previous page)

```
↪exiting.
            # For a non-blocking event loop replace `tcod.event.wait` with `tcod.event.
↪get`.
            for event in tcod.event.wait():
                context.convert_event(event)  # Sets tile coordinates for mouse events.
                print(event)  # Print event names and attributes.
                if isinstance(event, tcod.event.Quit):
                    raise SystemExit()
        # The window will be closed after the above with-block exits.


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

6.2 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 python
import tcod.context
import tcod.event

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 size based on window␣
↪resolution and tile size.
            console.print(0, 0, "Hello World")
            context.present(console, integer_scaling=True)
```

(continues on next page)

(continued from previous page)

```
            for event in tcod.event.wait():
                context.convert_event(event)  # Sets tile coordinates for mouse events.
                print(event)  # Print event names and attributes.
                if isinstance(event, tcod.event.Quit):
                    raise SystemExit()
                elif isinstance(event, tcod.event.WindowResized) and event.type ==
↪"WindowSizeChanged":
                    pass  # The next call to context.new_console may return a different␣
↪size.


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

CHAPTER

SEVEN

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.

7.1 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
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

continues on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
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
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

continues on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
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
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

continues on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
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	0x2302	'⌂'	HOUSE
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
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

continues on next page

Table 1 – continued from previous page

Tile Index	Unicode	String	Name
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
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

continues on next page

Table 1 – continued from previous page

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

7.2 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
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

continues on next page

Table 2 – continued from previous page

Tile Index	Unicode	String	Name
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
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'	

continues on next page

Table 2 – continued from previous page

Tile Index	Unicode	String	Name
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
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

continues on next page

Table 2 – continued from previous page

Tile Index	Unicode	String	Name
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

EIGHT

BINARY SPACE PARTITIONING TCOD.BSP

Libtcod's 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

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.

`__repr__`() → str

Provide a useful readout when printed.

`contains`(*x: int*, *y: int*) → bool

Return 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*) → *BSP* | None

Return the deepest node which contains these coordinates.

Returns
BSP object or None.

`in_order`() → Iterator[*BSP*]

Iterate over this BSP's hierarchy in order.

New in version 8.3.

`inverted_level_order`() → Iterator[*BSP*]

Iterate over this BSP's hierarchy in inverse level order.

New in version 8.3.

`level_order`() → Iterator[*BSP*]

Iterate over this BSP's hierarchy in level order.

New in version 8.3.

`post_order`() → Iterator[*BSP*]

Iterate over this BSP's hierarchy in post order.

New in version 8.3.

`pre_order`() → Iterator[*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*) – If True then the sub-partition is split into an upper and bottom half.
- **position** (*int*) – The position of where to put the divider relative to the current node.

`split_recursive`(*depth: int*, *min_width: int*, *min_height: int*, *max_horizontal_ratio: float*, *max_vertical_ratio: float*, *seed:* Random | *None* = *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[*BSP*]

Iterate over this BSP's hierarchy in pre order.

Deprecated since version 2.3: Use *pre_order* instead.

CHAPTER

NINE

TILE DRAWING/PRINTING `TCOD.CONSOLE`

Libtcod tile-based Consoles and printing functions.

Libtcod consoles are a strictly tile-based representation of colored glyphs/tiles. 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: Literal['C', 'F'] = 'C'*, *buffer: NDArray[Any] | None = 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 arranged. With the default setting of *"C"* you use [y, x] to index a consoles arrays such as `Console.rgb`. *order="F"* will swap the first two axes which allows for more intuitive *[x, y]* indexing. To be consistent you may have to do this for every NumPy array to create.

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

(continues on next page)

(continued from previous page)

```
xxxxxxxxxxxxxxxxxxx
____________________>
```

New in version 8.5.

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

__bool__() → bool

Return False if this is the root console.

This mimics libtcodpy behavior.

__enter__() → *Console*

Return 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 *libtcodpy.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:

libtcodpy.console_init_root

__exit__(*_: *object*) → None

Close 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:* 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: tuple[int, int, int] | None = 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 value.
- **bg_alpha** (*float*) – Background color alpha value.

- **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 rearranged 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 `libtcodpy.console_init_root`.

New in version 11.11.

draw_frame(*x: int*, *y: int*, *width: int*, *height: int*, *title: str = ''*, *clear: bool = True*, *fg: tuple[int, int, int] | None = None*, *bg: tuple[int, int, int] | None = None*, *bg_blend: int = 1*, ***, *decoration: str | tuple[int, int, int, int, int, int, int, int, int] = '-| |└-'*) → 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.

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. Using the *title* parameter is discouraged since the style it uses is hard-coded into libtcod. You should print over the top or bottom border with `Console.print_box` using your own style.

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.

decoration is a sequence of glyphs to use for rendering the borders. This a str or tuple of int's with 9 items with the items arranged in row-major order. If a *decoration* is given then *title* can not be used because the style for *title* is hard-coded. You can easily print along the upper or lower border of the frame manually.

New in version 8.5.

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

Changed in version 12.6: Added *decoration* parameter.

Changed in version 13.0: *x* and *y* are now always used as an absolute position for negative values.

Example:

```
>>> console = tcod.console.Console(12, 6)
>>> console.draw_frame(x=0, y=0, width=3, height=3)
>>> console.draw_frame(x=3, y=0, width=3, height=3, decoration=" ")
>>> console.draw_frame(x=6, y=0, width=3, height=3, decoration="123456789")
>>> console.draw_frame(x=9, y=0, width=3, height=3, decoration="/-\\| |\\-/")
>>> console.draw_frame(x=0, y=3, width=12, height=3)
>>> console.print_box(x=0, y=3, width=12, height=1, string=" Title ",␣
↪alignment=tcod.CENTER)
1
>>> console.print_box(x=0, y=5, width=12, height=1, string="Lower├",␣
↪alignment=tcod.CENTER)
1
>>> print(console)
<─123/-\
 │ │ 456| |
 └─789\-/
 ─ Title ──
 │          │
 └─Lower├──>
```

draw_rect(*x: int*, *y: int*, *width: int*, *height: int*, *ch: int*, *fg: tuple[int, int, int] | None = None*, *bg: tuple[int, int, int] | None = 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.

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.

Changed in version 13.0: *x* and *y* are now always used as an absolute position for negative values.

draw_semigraphics(*pixels: ArrayLike | tcod.image.Image*, *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: tuple[int, int, int] | None = None*, *bg: tuple[int, int, int] | None = 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.

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.

Changed in version 13.0: *x* and *y* are now always used as an absolute position for negative values.

print_(*x: int*, *y: int*, *string: str*, *bg_blend: int = 13*, *alignment: int | None = 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: tuple[int, int, int] | None = None*, *bg: tuple[int, int, int] | None = 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.

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.

Changed in version 13.0: *x* and *y* are now always used as an absolute position for negative values.

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: int | None = 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: tuple[int, int, int] | None*) → 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 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.

property bg: NDArray[np.uint8]

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'`.

property buffer: NDArray[Any]

An array of this consoles raw tile data.

New in version 11.4.

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

property ch: NDArray[np.intc]

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'`.

property default_alignment: int

The default text alignment.

Type
int

property default_bg: tuple[int, int, int]

The default background color.

Type
Tuple[int, int, int]

property default_bg_blend: int

The default blending mode.

Type
int

property default_fg: tuple[int, int, int]

The default foreground color.

Type
Tuple[int, int, int]

property fg: NDArray[np.uint8]

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'`.

property height: int

The height of this Console.

property rgb: NDArray[Any]

An array of this consoles data without the alpha channel.

The axes of this array is affected by the *order* parameter given to initialize the console.

The *rgb_graphic* dtype can be used to make arrays compatible with this attribute that are independent of a *Console*.

Example

```
>>> con = tcod.console.Console(10, 2)
>>> BLUE, YELLOW, BLACK = (0, 0, 255), (255, 255, 0), (0, 0, 0)
>>> con.rgb[0, 0] = ord("@"), YELLOW, BLACK
>>> con.rgb[0, 0]
(64, [255, 255,   0], [0, 0, 0])
>>> con.rgb["bg"] = BLUE
>>> con.rgb[0, 0]
(64, [255, 255,   0], [  0,   0, 255])
```

New in version 12.3.

property rgba: NDArray[Any]

An array of this consoles raw tile data.

The axes of this array is affected by the *order* parameter given to initialize the console.

Example

```
>>> con = tcod.console.Console(10, 2)
>>> WHITE, BLACK = (255, 255, 255), (0, 0, 0)
>>> con.rgba[0, 0] = (
...     ord("X"),
...     (*WHITE, 255),
...     (*BLACK, 255),
... )
>>> con.rgba[0, 0]
(88, [255, 255, 255, 255], [  0,   0,   0, 255])
```

New in version 12.3.

property tiles: NDArray[Any]

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.

New in version 10.0.

Deprecated since version 12.3: Use *Console.rgba* instead.

property tiles2: NDArray[Any]

This name is deprecated in favour of *rgb*.

New in version 11.3.

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

property tiles_rgb: NDArray[Any]

An array of this consoles data without the alpha channel.

New in version 11.8.

Deprecated since version 12.3: Use *Console.rgb* instead.

property width: int

The width of this Console.

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.**load_xp**(*path: str | PathLike[str]*, *order: Literal['C', 'F'] = 'C'*) → tuple[*Console*, ...]

Load a REXPaint file as a tuple of consoles.

path is the name of the REXPaint file to load. Usually ending with *.xp*.

order is the memory order of the Console's array buffer, see *tcod.console.Console*.

New in version 12.4.

Example:

```
import numpy as np
import tcod.console
import tcod.tileset

path = "example.xp"  # REXPaint file with one layer.

# Load a REXPaint file with a single layer.
# The comma after console is used to unpack a single item tuple.
console, = tcod.console.load_xp(path, order="F")

# Convert tcod's Code Page 437 character mapping into a NumPy array.
CP437_TO_UNICODE = np.asarray(tcod.tileset.CHARMAP_CP437)

# Convert from REXPaint's CP437 encoding to Unicode in-place.
```

(continues on next page)

(continued from previous page)

```
console.ch[:] = CP437_TO_UNICODE[console.ch]

# Apply REXPaint's alpha key color.
KEY_COLOR = (255, 0, 255)
is_transparent = (console.rgb["bg"] == KEY_COLOR).all(axis=-1)
console.rgba[is_transparent] = (ord(" "), (0,), (0,))
```

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:

libtcodpy.console_init_root *libtcodpy.console_flush*

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

tcod.console.**save_xp**(*path: str | PathLike[str]*, *consoles: Iterable[*Console*]*, *compress_level: int = 9*) → None

Save tcod Consoles to a REXPaint file.

path is where to save the file.

consoles are the *tcod.console.Console* objects to be saved.

compress_level is the zlib compression level to be used.

Color alpha will be lost during saving.

Consoles will be saved as-is as much as possible. You may need to convert characters from Unicode to CP437 if you want to load the file in REXPaint.

New in version 12.4.

Example:

```
import numpy as np
import tcod.console
import tcod.tileset

console = tcod.console.Console(80, 24)  # Example console.

# Convert from Unicode to REXPaint's encoding.
# Required to load this console correctly in the REXPaint tool.

# Convert tcod's Code Page 437 character mapping into a NumPy array.
CP437_TO_UNICODE = np.asarray(tcod.tileset.CHARMAP_CP437)

# Initialize a Unicode-to-CP437 array.
# 0x20000 is the current full range of Unicode.
# fill_value=ord("?") means that "?" will be the result of any unknown codepoint.
UNICODE_TO_CP437 = np.full(0x20000, fill_value=ord("?"))
# Assign the CP437 mappings.
```

(continues on next page)

(continued from previous page)

```
UNICODE_TO_CP437[CP437_TO_UNICODE] = np.arange(len(CP437_TO_UNICODE))

# Convert from Unicode to CP437 in-place.
console.ch[:] = UNICODE_TO_CP437[console.ch]

# Convert console alpha into REXPaint's alpha key color.
KEY_COLOR = (255, 0, 255)
is_transparent = console.rgba["bg"][:, :, 3] == 0
console.rgb["bg"][is_transparent] = KEY_COLOR

tcod.console.save_xp("example.xp", [console])
```

tcod.console.**rgb_graphic = dtype([('ch', '<i4'), ('fg', 'u1', (3,)), ('bg', 'u1', (3,))])**

A NumPy `dtype` compatible with *Console.rgb*.

This dtype is: `np.dtype([("ch", np.intc), ("fg", "3B"), ("bg", "3B")])`

New in version 12.3.

tcod.console.**rgba_graphic = dtype([('ch', '<i4'), ('fg', 'u1', (4,)), ('bg', 'u1', (4,))])**

A NumPy `dtype` compatible with *Console.rgba*.

This dtype is: `np.dtype([("ch", np.intc), ("fg", "4B"), ("bg", "4B")])`

New in version 12.3.

CHAPTER

TEN

WINDOW MANAGEMENT `TCOD.CONTEXT`

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 *libtcodpy.console_init_root*, *libtcodpy.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 *libtcodpy.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 expect deprecated functions from `libtcodpy` to no longer work correctly. Those functions rely on a global console or tileset which doesn't exists with contexts. Also `libtcodpy` event functions will no longer return tile coordinates for the mouse.

New programs not using `libtcodpy` can ignore this warning.

New in version 11.12.

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

Context manager for libtcod context objects.

Use *tcod.context.new* to create a new context.

__enter__() → *Context*

Enter this context which will close on exiting.

__exit__(**_: object*) → None

Automatically close on the context on exit.

__reduce__() → NoReturn

Contexts can not be pickled, so this class will raise `pickle.PicklingError`.

change_tileset(*tileset:* Tileset | *None*) → None

Change the active tileset used by this context.

The new tileset will take effect on the next call to *present*. Contexts not using a renderer with an emulated terminal will be unaffected by this method.

This does not do anything to resize the window, keep this in mind if the tileset as a differing tile size. Access the window with *sdl_window* to resize it manually, if needed.

Using this method only one tileset is active per-frame. See *tcod.render* if you want to renderer with multiple tilesets in a single frame.

close() → None

Close this context, closing any windows opened by this context.

Afterwards doing anything with this instance other than closing it again is invalid.

convert_event(*event: _Event*) → _Event

Return an event with mouse pixel coordinates converted into tile coordinates.

Example:

```
context: tcod.context.Context
for event in tcod.event.get():
    event_tile = context.convert_event(event)
    if isinstance(event, tcod.event.MouseMotion):
        # Events start with pixel coordinates and motion.
        print(f"Pixels: {event.position=}, {event.motion=}")
    if isinstance(event_tile, tcod.event.MouseMotion):
        # Tile coordinates are used in the returned event.
        print(f"Tiles: {event_tile.position=}, {event_tile.motion=}")
```

Changed in version 15.0: Now returns a new event with the coordinates converted into tiles.

new_console(*min_columns: int = 1*, *min_rows: int = 1*, *magnification: float = 1.0*, *order: Literal['C', 'F'] = 'C'*) → *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

Example:

```
scale = 1  # Tile size scale.  This example uses integers but floating point␣
↪numbers are also valid.
context = tcod.context.new()
while True:
    # Create a cleared, dynamically-sized console for each frame.
    console = context.new_console(magnification=scale)
    # This printed output will wrap if the window is shrunk.
    console.print_box(0, 0, console.width, console.height, "Hello world")
    # Use integer_scaling to prevent subpixel distortion.
```

(continues on next page)

(continued from previous page)

```
    # This may add padding around the rendered console.
    context.present(console, integer_scaling=True)
    for event in tcod.event.wait():
        if isinstance(event, tcod.event.Quit):
            raise SystemExit()
        elif isinstance(event, tcod.event.MouseWheel):
            # Use the mouse wheel to change the rendered tile size.
            scale = max(1, scale + event.y)
```

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:* 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: str | None = None*) → None

Save a screen-shot to the given file path.

property renderer_type: int

Return the libtcod renderer type used by this context.

property sdl_atlas: *SDLTilesetAtlas* | None

Return a *tcod.render.SDLTilesetAtlas* referencing libtcod's SDL texture atlas if it exists.

New in version 13.5.

property sdl_renderer: *Renderer* | None

Return a *tcod.sdl.render.Renderer* referencing this contexts SDL renderer if it exists.

New in version 13.4.

property sdl_window: *Window* | None

Return a *tcod.sdl.video.Window* referencing this contexts SDL window if it exists.

Example:

```
import tcod
import tcod.sdl.video

def toggle_fullscreen(context: tcod.context.Context) -> None:
    """Toggle a context window between fullscreen and windowed modes."""
    window = context.sdl_window
    if not window:
        return
    if window.fullscreen:
        window.fullscreen = False
    else:
        window.fullscreen = tcod.sdl.video.WindowFlags.FULLSCREEN_DESKTOP
```

New in version 13.4.

property sdl_window_p: Any

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: int* | *None* = *None*, *y: int* | *None* = *None*, *width: int* | *None* = *None*, *height: int* | *None* = *None*, *columns: int* | *None* = *None*, *rows: int* | *None* = *None*, *renderer: int* | *None* = *None*, *tileset:* Tileset | *None* = *None*, *vsync: bool* = *True*, *sdl_window_flags: int* | *None* = *None*, *title: str* | *None* = *None*, *argv: Iterable[str]* | *None* = *None*, *console:* Console | *None* = *None*) → *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.

console automatically fills in the *columns* and *rows* parameters from an existing `tcod.console.Console` instance.

Providing no size information at all is also acceptable.

renderer now does nothing and should not be set. It may be removed in the future.

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.

Changed in version 13.2: Added the *console* parameter.

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

Create a new context with the desired console size.

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

`tcod.context.new_window`(*width: int*, *height: int*, ***, *renderer: int | None = None*, *tileset: Tileset | None = None*, *vsync: bool = True*, *sdl_window_flags: int | None = None*, *title: str | None = None*) → *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.RENDERER_OPENGL = 1`

A renderer for older versions of OpenGL.

Should support OpenGL 1 and GLES 1

`tcod.context.RENDERER_OPENGL2 = 4`

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 = 2`

Same as RENDERER_SDL2, but forces SDL2 into software mode.

tcod.context.**RENDERER_SDL2 = 3**

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

tcod.context.**RENDERER_XTERM = 5**

A renderer targeting modern terminals with 24-bit color support.

This is an experimental renderer with partial support for XTerm and SSH. This will work best on those terminals.

Terminal inputs and events will be passed to SDL's event system.

There is poor support for ANSI escapes on Windows 10. It is not recommended to use this renderer on Windows.

New in version 13.3.

tcod.context.**SDL_WINDOW_ALLOW_HIGHDPI = 8192**

High DPI mode, see the SDL documentation.

tcod.context.**SDL_WINDOW_BORDERLESS = 16**

Window has no decorative border.

tcod.context.**SDL_WINDOW_FULLSCREEN = 1**

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 = 4097**

A borderless fullscreen window at the desktop resolution.

tcod.context.**SDL_WINDOW_HIDDEN = 8**

Window is hidden.

tcod.context.**SDL_WINDOW_INPUT_GRABBED = 256**

Window has grabbed the input.

tcod.context.**SDL_WINDOW_MAXIMIZED = 128**

Window is maximized.

tcod.context.**SDL_WINDOW_MINIMIZED = 64**

Window is minimized.

tcod.context.**SDL_WINDOW_RESIZABLE = 32**

Window can be resized.

CHAPTER

ELEVEN

SDL2 EVENT HANDLING `TCOD.EVENT`

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

Many event constants are derived directly from SDL. For example: `tcod.event.KeySym.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.

Example:

```
import tcod

KEY_COMMANDS = {
    tcod.event.KeySym.UP: "move N",
    tcod.event.KeySym.DOWN: "move S",
    tcod.event.KeySym.LEFT: "move W",
    tcod.event.KeySym.RIGHT: "move E",
}

context = tcod.context.new()
while True:
    console = context.new_console()
    context.present(console, integer_scaling=True)
    for event in tcod.event.wait():
        context.convert_event(event)  # Adds tile coordinates to mouse events.
        if isinstance(event, tcod.event.Quit):
            print(event)
            raise SystemExit()
        elif isinstance(event, tcod.event.KeyDown):
            print(event)  # Prints the Scancode and KeySym enums for this event.
            if event.sym in KEY_COMMANDS:
                print(f"Command: {KEY_COMMANDS[event.sym]}")
        elif isinstance(event, tcod.event.MouseButtonDown):
            print(event)  # Prints the mouse button constant names for this event.
        elif isinstance(event, tcod.event.MouseMotion):
            print(event)  # Prints the mouse button mask bits in a readable format.
        else:
            print(event)  # Print any unhandled events.
```

Python 3.10 introduced match statements which can be used to dispatch events more gracefully:

Example:

```
import tcod

KEY_COMMANDS = {
    tcod.event.KeySym.UP: "move N",
    tcod.event.KeySym.DOWN: "move S",
    tcod.event.KeySym.LEFT: "move W",
    tcod.event.KeySym.RIGHT: "move E",
}

context = tcod.context.new()
while True:
    console = context.new_console()
    context.present(console, integer_scaling=True)
    for event in tcod.event.wait():
        context.convert_event(event)  # Adds tile coordinates to mouse events.
        match event:
            case tcod.event.Quit():
                raise SystemExit()
            case tcod.event.KeyDown(sym) if sym in KEY_COMMANDS:
                print(f"Command: {KEY_COMMANDS[sym]}")
            case tcod.event.KeyDown(sym=sym, scancode=scancode, mod=mod, repeat=repeat):
                print(f"KeyDown: {sym=}, {scancode=}, {mod=}, {repeat=}")
            case tcod.event.MouseButtonDown(button=button, pixel=pixel, tile=tile):
                print(f"MouseButtonDown: {button=}, {pixel=}, {tile=}")
            case tcod.event.MouseMotion(pixel=pixel, pixel_motion=pixel_motion,␣
↪tile=tile, tile_motion=tile_motion):
                print(f"MouseMotion: {pixel=}, {pixel_motion=}, {tile=}, {tile_motion=}")
            case tcod.event.Event() as event:
                print(event)  # Show any unhandled events.
```

New in version 8.4.

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

A 2D position used for events with mouse coordinates.

See also:

MouseMotion *MouseButtonDown* *MouseButtonUp*

x: int

A pixel or tile coordinate starting with zero as the left-most position.

y: int

A pixel or tile coordinate starting with zero as the top-most position.

class tcod.event.**Event**(*type: str | None = 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*) → *Event*

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

class tcod.event.**Quit**(*type: str | None = 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*) → *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*)

Base keyboard event.

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
Scancode

sym

The keyboard symbol.

Type
KeySym

mod

A bitmask of the currently held modifier keys.

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

Type
Modifier

repeat

True if this event exists because of key repeat.

Type
bool

Changed in version 12.5: *scancode*, *sym*, and *mod* now use their respective enums.

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**(*position: tuple[int, int] = (0, 0)*, *motion: tuple[int, int] = (0, 0)*, *tile: tuple[int, int] | None = (0, 0)*, *tile_motion: tuple[int, int] | None = (0, 0)*, *state: int = 0*)

Mouse motion event.

type

Always "MOUSEMOTION".

Type
str

position

The pixel coordinates of the mouse.

Type
Point

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

Changed in version 15.0: Renamed *pixel* attribute to *position*. Renamed *pixel_motion* attribute to *motion*.

classmethod **from_sdl_event**(*sdl_event: Any*) → *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] | None = (0, 0)*, *button: int = 0*)

Mouse button event.

type

Will be “MOUSEBUTTONDOWN” or “MOUSEBUTTONUP”, depending on the event.

Type
str

position

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] | None = (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] | None = (0, 0)*, *button: int = 0*)

Same as MouseButtonEvent but with `type="MouseButtonUp"`.

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

Mouse wheel event.

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*) → *MouseWheel*

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

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

SDL text input event.

type

Always "TEXTINPUT".

Type
str

text

A Unicode string with the input.

Type
str

classmethod from_sdl_event(*sdl_event: Any*) → *TextInput*

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

class tcod.event.**WindowEvent**(*type: str | None = None*)

A window event.

type: Final[Literal['WindowShown', 'WindowHidden', 'WindowExposed', 'WindowMoved', 'WindowResized', 'WindowSizeChanged', 'WindowMinimized', 'WindowMaximized', 'WindowRestored', 'WindowEnter', 'WindowLeave', 'WindowFocusGained', 'WindowFocusLost', 'WindowClose', 'WindowTakeFocus', 'WindowHitTest']]

The current window event. This can be one of various options.

classmethod from_sdl_event(*sdl_event: Any*) → *WindowEvent | Undefined*

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

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

Window moved event.

x

Movement on the x-axis.

Type
int

y

Movement on the y-axis.

Type
int

type: Final[Literal['WINDOWMOVED']]

Always “WINDOWMOVED”.

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

Window resized event.

width

The current width of the window.

Type
int

height

The current height of the window.

Type
int

type: Final[Literal['WindowResized', 'WindowSizeChanged']]

WindowResized” or “WindowSizeChanged

class tcod.event.**JoystickEvent**(*type: str*, *which: int*)

A base class for joystick events.

New in version 13.8.

which

The ID of the joystick this event is for.

class tcod.event.**JoystickAxis**(*type: str*, *which: int*, *axis: int*, *value: int*)

When a joystick axis changes in value.

New in version 13.8.

See also:

tcod.sdl.joystick

which: int

The ID of the joystick this event is for.

axis

The index of the changed axis.

value

The raw value of the axis in the range -32768 to 32767.

classmethod from_sdl_event(*sdl_event: Any*) → *JoystickAxis*

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

class tcod.event.**JoystickBall**(*type: str*, *which: int*, *ball: int*, *dx: int*, *dy: int*)

When a joystick ball is moved.

New in version 13.8.

See also:

tcod.sdl.joystick

which: *int*

The ID of the joystick this event is for.

ball

The index of the moved ball.

dx

The X motion of the ball.

dy

The Y motion of the ball.

classmethod from_sdl_event(*sdl_event: Any*) → *JoystickBall*

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

class tcod.event.**JoystickHat**(*type: str*, *which: int*, *x: Literal[-1, 0, 1]*, *y: Literal[-1, 0, 1]*)

When a joystick hat changes direction.

New in version 13.8.

See also:

tcod.sdl.joystick

which: *int*

The ID of the joystick this event is for.

x

The new X direction of the hat.

y

The new Y direction of the hat.

classmethod from_sdl_event(*sdl_event: Any*) → *JoystickHat*

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

class tcod.event.**JoystickButton**(*type: str*, *which: int*, *button: int*)

When a joystick button is pressed or released.

New in version 13.8.

Example:

```
for event in tcod.event.get():
    match event:
        case JoystickButton(which=which, button=button, pressed=True):
            print(f"Pressed {button=} on controller {which}.")
        case JoystickButton(which=which, button=button, pressed=False):
            print(f"Released {button=} on controller {which}.")
```

which: *int*

The ID of the joystick this event is for.

button

The index of the button this event is for.

property pressed: bool

True if the joystick button has been pressed, False when the button was released.

classmethod from_sdl_event(*sdl_event: Any*) → *JoystickButton*

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

class tcod.event.**JoystickDevice**(*type: str*, *which: int*)

An event for when a joystick is added or removed.

New in version 13.8.

Example:

```
joysticks: set[tcod.sdl.joystick.Joystick] = {}
for event in tcod.event.get():
    match event:
        case tcod.event.JoystickDevice(type="JOYDEVICEADDED", joystick=new_
↪joystick):
            joysticks.add(new_joystick)
        case tcod.event.JoystickDevice(type="JOYDEVICEREMOVED", joystick=joystick):
            joysticks.remove(joystick)
```

which: int

When type="JOYDEVICEADDED" this is the device ID. When type="JOYDEVICEREMOVED" this is the instance ID.

classmethod from_sdl_event(*sdl_event: Any*) → *JoystickDevice*

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

class tcod.event.**ControllerEvent**(*type: str*, *which: int*)

Base class for controller events.

New in version 13.8.

which

The ID of the joystick this event is for.

property controller: *GameController*

The *GameController* for this event.

class tcod.event.**ControllerAxis**(*type: str*, *which: int*, *axis:* ControllerAxis, *value: int*)

When a controller axis is moved.

New in version 13.8.

axis

Which axis is being moved. One of *ControllerAxis*.

value

The new value of this events axis.

This will be -32768 to 32767 for all axes except for triggers which are 0 to 32767 instead.

classmethod from_sdl_event(*sdl_event: Any*) → *ControllerAxis*

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

class tcod.event.**ControllerButton**(*type: str*, *which: int*, *button:* ControllerButton, *pressed: bool*)

When a controller button is pressed or released.

New in version 13.8.

button

The button for this event. One of *ControllerButton*.

pressed

True if the button was pressed, False if it was released.

classmethod from_sdl_event(*sdl_event: Any*) → *ControllerButton*

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

class tcod.event.**ControllerDevice**(*type: str*, *which: int*)

When a controller is added, removed, or remapped.

New in version 13.8.

classmethod from_sdl_event(*sdl_event: Any*) → *ControllerDevice*

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

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*) → *Undefined*

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

tcod.event.**get_mouse_state**() → MouseState

Return the current state of the mouse.

New in version 9.3.

tcod.event.**add_watch**(*callback: _EventCallback*) → _EventCallback

Add a callback for watching events.

This function can be called with the callback to register, or be used as a decorator.

Callbacks added as event watchers can later be removed with *tcod.event.remove_watch*.

> **Warning:** How uncaught exceptions in a callback are handled is not currently defined by tcod. They will likely be handled by sys.unraisablehook. This may be later changed to pass the exception to a *tcod.event.get* or *tcod.event.wait* call.

Parameters

callback (*Callable[[Event], None]*) – A function which accepts *Event* parameters.

Example:

```
import tcod.event

@tcod.event.add_watch
def handle_events(event: tcod.event.Event) -> None:
    if isinstance(event, tcod.event.KeyDown):
        print(event)
```

New in version 13.4.

tcod.event.**remove_watch**(*callback: Callable[[Event], None]*) → None

Remove a callback as an event watcher.

Parameters

callback (*Callable[[Event], None]*) – A function which has been previously registered with *tcod.event.add_watch*.

New in version 13.4.

class tcod.event.**EventDispatch**

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 hints at the return value of *dispatch* and the *ev_** methods.

Example:

```
import tcod

MOVE_KEYS = {  # key_symbol: (x, y)
    # Arrow keys.
    tcod.event.KeySym.LEFT: (-1, 0),
    tcod.event.KeySym.RIGHT: (1, 0),
    tcod.event.KeySym.UP: (0, -1),
    tcod.event.KeySym.DOWN: (0, 1),
    tcod.event.KeySym.HOME: (-1, -1),
    tcod.event.KeySym.END: (-1, 1),
    tcod.event.KeySym.PAGEUP: (1, -1),
    tcod.event.KeySym.PAGEDOWN: (1, 1),
    tcod.event.KeySym.PERIOD: (0, 0),
    # Numpad keys.
    tcod.event.KeySym.KP_1: (-1, 1),
    tcod.event.KeySym.KP_2: (0, 1),
    tcod.event.KeySym.KP_3: (1, 1),
    tcod.event.KeySym.KP_4: (-1, 0),
    tcod.event.KeySym.KP_5: (0, 0),
    tcod.event.KeySym.KP_6: (1, 0),
    tcod.event.KeySym.KP_7: (-1, -1),
    tcod.event.KeySym.KP_8: (0, -1),
    tcod.event.KeySym.KP_9: (1, -1),
    tcod.event.KeySym.CLEAR: (0, 0),  # Numpad `clear` key.
    # Vi Keys.
    tcod.event.KeySym.h: (-1, 0),
    tcod.event.KeySym.j: (0, 1),
    tcod.event.KeySym.k: (0, -1),
    tcod.event.KeySym.l: (1, 0),
    tcod.event.KeySym.y: (-1, -1),
    tcod.event.KeySym.u: (1, -1),
    tcod.event.KeySym.b: (-1, 1),
    tcod.event.KeySym.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.KeySym.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 = libtcodpy.console_init_root(80, 60)
state = State()
while True:
    libtcodpy.console_flush()
    for event in tcod.event.wait():
        state.dispatch(event)
```

dispatch(*event: Any*) → T | None

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_quit(*event:* Quit) → T | None

Called when the termination of the program is requested.

ev_keydown(*event:* KeyDown) → T | None

Called when a keyboard key is pressed or repeated.

ev_keyup(*event:* KeyUp) → T | None

Called when a keyboard key is released.

ev_mousemotion(*event:* MouseMotion) → T | None

Called when the mouse is moved.

ev_mousebuttondown(*event:* MouseButtonDown) → T | None

Called when a mouse button is pressed.

ev_mousebuttonup(*event:* MouseButtonUp) → T | None

Called when a mouse button is released.

ev_mousewheel(*event:* MouseWheel) → T | None

Called when the mouse wheel is scrolled.

ev_textinput(*event:* TextInput) → T | None

Called to handle Unicode input.

ev_windowshown(*event:* WindowEvent) → T | None

Called when the window is shown.

ev_windowhidden(*event:* WindowEvent) → T | None

Called when the window is hidden.

ev_windowexposed(*event:* WindowEvent) → T | None

Called when a window is exposed, and needs to be refreshed.

This usually means a call to *libtcodpy.console_flush* is necessary.

ev_windowmoved(*event:* WindowMoved) → T | None

Called when the window is moved.

ev_windowresized(*event:* WindowResized) → T | None

Called when the window is resized.

ev_windowsizechanged(*event:* WindowResized) → T | None

Called when the system or user changes the size of the window.

ev_windowminimized(*event:* WindowEvent) → T | None

Called when the window is minimized.

ev_windowmaximized(*event:* WindowEvent) → T | None

Called when the window is maximized.

ev_windowrestored(*event:* WindowEvent) → T | None
Called when the window is restored.

ev_windowenter(*event:* WindowEvent) → T | None
Called when the window gains mouse focus.

ev_windowleave(*event:* WindowEvent) → T | None
Called when the window loses mouse focus.

ev_windowfocusgained(*event:* WindowEvent) → T | None
Called when the window gains keyboard focus.

ev_windowfocuslost(*event:* WindowEvent) → T | None
Called when the window loses keyboard focus.

ev_windowclose(*event:* WindowEvent) → T | None
Called when the window manager requests the window to be closed.

ev_joyaxismotion(*event:* JoystickAxis) → T | None
Called when a joystick analog is moved.

New in version 13.8.

ev_joyballmotion(*event:* JoystickBall) → T | None
Called when a joystick ball is moved.

New in version 13.8.

ev_joyhatmotion(*event:* JoystickHat) → T | None
Called when a joystick hat is moved.

New in version 13.8.

ev_joybuttondown(*event:* JoystickButton) → T | None
Called when a joystick button is pressed.

New in version 13.8.

ev_joybuttonup(*event:* JoystickButton) → T | None
Called when a joystick button is released.

New in version 13.8.

ev_joydeviceadded(*event:* JoystickDevice) → T | None
Called when a joystick is added.

New in version 13.8.

ev_joydeviceremoved(*event:* JoystickDevice) → T | None
Called when a joystick is removed.

New in version 13.8.

ev_controlleraxismotion(*event:* ControllerAxis) → T | None
Called when a controller analog is moved.

New in version 13.8.

ev_controllerbuttondown(*event:* ControllerButton) → T | None
Called when a controller button is pressed.

New in version 13.8.

ev_controllerbuttonup(*event:* ControllerButton) → T | None

Called when a controller button is released.

New in version 13.8.

ev_controllerdeviceadded(*event:* ControllerDevice) → T | None

Called when a standard controller is added.

New in version 13.8.

ev_controllerdeviceremoved(*event:* ControllerDevice) → T | None

Called when a standard controller is removed.

New in version 13.8.

ev_controllerdeviceremapped(*event:* ControllerDevice) → T | None

Called when a standard controller is remapped.

New in version 13.8.

tcod.event.**get_keyboard_state**() → NDArray[np.bool_]

Return a boolean array with the current keyboard state.

Index this array with a scancode. The value will be True if the key is currently held.

Example:

```
state = tcod.event.get_keyboard_state()

# Get a WASD movement vector:
x = int(state[tcod.event.Scancode.D]) - int(state[tcod.event.Scancode.A])
y = int(state[tcod.event.Scancode.S]) - int(state[tcod.event.Scancode.W])

# Key with 'z' glyph is held:
is_z_held = state[tcod.event.KeySym.z.scancode]
```

New in version 12.3.

tcod.event.**get_modifier_state**() → *Modifier*

Return a bitmask of the active keyboard modifiers.

New in version 12.3.

tcod.event.**__getattr__**(*name: str*) → int

Migrate deprecated access of event constants.

11.1 Getting events

The primary way to capture events is with the *tcod.event.get* and *tcod.event.wait* functions. These functions return events in a loop until the internal event queue is empty. Use isinstance(), *tcod.event.EventDispatch*, or match statements (introduced in Python 3.10) to determine which event was returned.

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.)

tcod.event.**wait**(*timeout: float | None = 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*.

This function is useful for simple games with little to no animations. The following example sleeps whenever no events are queued:

Example:

```
context: tcod.context.Context  # Context object initialized earlier.
while True:  # Main game-loop.
    console: tcod.console.Console  # Console used for rendering.
    ...  # Render the frame to `console` and then:
    context.present(console)  # Show the console to the display.
    # The ordering to draw first before waiting for events is important.
    for event in tcod.event.wait():  # Sleeps until the next events exist.
        ...  # All events are handled at once before the next frame.
```

See *tcod.event.get* examples for how different events are handled.

11.2 Keyboard Enums

- *KeySym*: Keys based on their glyph.
- *Scancode*: Keys based on their physical location.
- *Modifier*: Keyboard modifier keys.

class tcod.event.**KeySym**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Keyboard constants based on their symbol.

These names are derived from SDL except for the numbers which are prefixed with `N` (since raw numbers can not be a Python name.)

New in version 12.3.

UNKNOWN	0
BACKSPACE	8
TAB	9
RETURN	13
ESCAPE	27
SPACE	32
EXCLAIM	33
QUOTEDBL	34
HASH	35
DOLLAR	36
PERCENT	37
AMPERSAND	38
QUOTE	39
LEFTPAREN	40

continues on next page

Table 1 – continued from previous page

RIGHTPAREN	41
ASTERISK	42
PLUS	43
COMMA	44
MINUS	45
PERIOD	46
SLASH	47
N0	48
N1	49
N2	50
N3	51
N4	52
N5	53
N6	54
N7	55
N8	56
N9	57
COLON	58
SEMICOLON	59
LESS	60
EQUALS	61
GREATER	62
QUESTION	63
AT	64
LEFTBRACKET	91
BACKSLASH	92
RIGHTBRACKET	93
CARET	94
UNDERSCORE	95
BACKQUOTE	96
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118

continues on next page

Table 1 – continued from previous page

w	119
x	120
y	121
z	122
DELETE	127
SCANCODE_MASK	1073741824
CAPSLOCK	1073741881
F1	1073741882
F2	1073741883
F3	1073741884
F4	1073741885
F5	1073741886
F6	1073741887
F7	1073741888
F8	1073741889
F9	1073741890
F10	1073741891
F11	1073741892
F12	1073741893
PRINTSCREEN	1073741894
SCROLLLOCK	1073741895
PAUSE	1073741896
INSERT	1073741897
HOME	1073741898
PAGEUP	1073741899
END	1073741901
PAGEDOWN	1073741902
RIGHT	1073741903
LEFT	1073741904
DOWN	1073741905
UP	1073741906
NUMLOCKCLEAR	1073741907
KP_DIVIDE	1073741908
KP_MULTIPLY	1073741909
KP_MINUS	1073741910
KP_PLUS	1073741911
KP_ENTER	1073741912
KP_1	1073741913
KP_2	1073741914
KP_3	1073741915
KP_4	1073741916
KP_5	1073741917
KP_6	1073741918
KP_7	1073741919
KP_8	1073741920
KP_9	1073741921
KP_0	1073741922
KP_PERIOD	1073741923
APPLICATION	1073741925
POWER	1073741926
KP_EQUALS	1073741927
F13	1073741928

continues on next page

Table 1 – continued from previous page

F14	1073741929
F15	1073741930
F16	1073741931
F17	1073741932
F18	1073741933
F19	1073741934
F20	1073741935
F21	1073741936
F22	1073741937
F23	1073741938
F24	1073741939
EXECUTE	1073741940
HELP	1073741941
MENU	1073741942
SELECT	1073741943
STOP	1073741944
AGAIN	1073741945
UNDO	1073741946
CUT	1073741947
COPY	1073741948
PASTE	1073741949
FIND	1073741950
MUTE	1073741951
VOLUMEUP	1073741952
VOLUMEDOWN	1073741953
KP_COMMA	1073741957
KP_EQUALSAS400	1073741958
ALTERASE	1073741977
SYSREQ	1073741978
CANCEL	1073741979
CLEAR	1073741980
PRIOR	1073741981
RETURN2	1073741982
SEPARATOR	1073741983
OUT	1073741984
OPER	1073741985
CLEARAGAIN	1073741986
CRSEL	1073741987
EXSEL	1073741988
KP_00	1073742000
KP_000	1073742001
THOUSANDSSEPARATOR	1073742002
DECIMALSEPARATOR	1073742003
CURRENCYUNIT	1073742004
CURRENCYSUBUNIT	1073742005
KP_LEFTPAREN	1073742006
KP_RIGHTPAREN	1073742007
KP_LEFTBRACE	1073742008
KP_RIGHTBRACE	1073742009
KP_TAB	1073742010
KP_BACKSPACE	1073742011
KP_A	1073742012

continues on next page

Table 1 – continued from previous page

KP_B	1073742013
KP_C	1073742014
KP_D	1073742015
KP_E	1073742016
KP_F	1073742017
KP_XOR	1073742018
KP_POWER	1073742019
KP_PERCENT	1073742020
KP_LESS	1073742021
KP_GREATER	1073742022
KP_AMPERSAND	1073742023
KP_DBLAMPERSAND	1073742024
KP_VERTICALBAR	1073742025
KP_DBLVERTICALBAR	1073742026
KP_COLON	1073742027
KP_HASH	1073742028
KP_SPACE	1073742029
KP_AT	1073742030
KP_EXCLAM	1073742031
KP_MEMSTORE	1073742032
KP_MEMRECALL	1073742033
KP_MEMCLEAR	1073742034
KP_MEMADD	1073742035
KP_MEMSUBTRACT	1073742036
KP_MEMMULTIPLY	1073742037
KP_MEMDIVIDE	1073742038
KP_PLUSMINUS	1073742039
KP_CLEAR	1073742040
KP_CLEARENTRY	1073742041
KP_BINARY	1073742042
KP_OCTAL	1073742043
KP_DECIMAL	1073742044
KP_HEXADECIMAL	1073742045
LCTRL	1073742048
LSHIFT	1073742049
LALT	1073742050
LGUI	1073742051
RCTRL	1073742052
RSHIFT	1073742053
RALT	1073742054
RGUI	1073742055
MODE	1073742081
AUDIONEXT	1073742082
AUDIOPREV	1073742083
AUDIOSTOP	1073742084
AUDIOPLAY	1073742085
AUDIOMUTE	1073742086
MEDIASELECT	1073742087
WWW	1073742088
MAIL	1073742089
CALCULATOR	1073742090
COMPUTER	1073742091

continues on next page

Table 1 – continued from previous page

AC_SEARCH	1073742092
AC_HOME	1073742093
AC_BACK	1073742094
AC_FORWARD	1073742095
AC_STOP	1073742096
AC_REFRESH	1073742097
AC_BOOKMARKS	1073742098
BRIGHTNESSDOWN	1073742099
BRIGHTNESSUP	1073742100
DISPLAYSWITCH	1073742101
KBDILLUMTOGGLE	1073742102
KBDILLUMDOWN	1073742103
KBDILLUMUP	1073742104
EJECT	1073742105
SLEEP	1073742106

__repr__() → str

Return the fully qualified name of this enum.

property keysym: *KeySym*

Return a keycode from a scancode.

Returns itself since it is already a *KeySym*.

See also:

Scancode.keysym

property label: str

A human-readable name of a keycode.

Returns "" if the keycode doesn't have a name.

Be sure not to confuse this with `.name`, which will return the enum name rather than the human-readable name.

Example:

```
>>> tcod.event.KeySym.F1.label
'F1'
>>> tcod.event.KeySym.BACKSPACE.label
'Backspace'
```

property scancode: *Scancode*

Return a scancode from a keycode.

Based on the current keyboard layout.

class tcod.event.**Scancode**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

A Scancode represents the physical location of a key.

For example the scan codes for WASD remain in the same physical location regardless of the actual keyboard layout.

These names are derived from SDL except for the numbers which are prefixed with `N` (since raw numbers can not be a Python name.)

New in version 12.3.

UNKNOWN	0
A	4
B	5
C	6
D	7
E	8
F	9
G	10
H	11
I	12
J	13
K	14
L	15
M	16
N	17
O	18
P	19
Q	20
R	21
S	22
T	23
U	24
V	25
W	26
X	27
Y	28
Z	29
N1	30
N2	31
N3	32
N4	33
N5	34
N6	35
N7	36
N8	37
N9	38
N0	39
RETURN	40
ESCAPE	41
BACKSPACE	42
TAB	43
SPACE	44
MINUS	45
EQUALS	46
LEFTBRACKET	47
RIGHTBRACKET	48
BACKSLASH	49
NONUSHASH	50
SEMICOLON	51
APOSTROPHE	52

continues on next page

Table 2 – continued from previous page

GRAVE	53
COMMA	54
PERIOD	55
SLASH	56
CAPSLOCK	57
F1	58
F2	59
F3	60
F4	61
F5	62
F6	63
F7	64
F8	65
F9	66
F10	67
F11	68
F12	69
PRINTSCREEN	70
SCROLLLOCK	71
PAUSE	72
INSERT	73
HOME	74
PAGEUP	75
DELETE	76
END	77
PAGEDOWN	78
RIGHT	79
LEFT	80
DOWN	81
UP	82
NUMLOCKCLEAR	83
KP_DIVIDE	84
KP_MULTIPLY	85
KP_MINUS	86
KP_PLUS	87
KP_ENTER	88
KP_1	89
KP_2	90
KP_3	91
KP_4	92
KP_5	93
KP_6	94
KP_7	95
KP_8	96
KP_9	97
KP_0	98
KP_PERIOD	99
NONUSBACKSLASH	100
APPLICATION	101
POWER	102
KP_EQUALS	103
F13	104

continues on next page

Table 2 – continued from previous page

F14	105
F15	106
F16	107
F17	108
F18	109
F19	110
F20	111
F21	112
F22	113
F23	114
F24	115
EXECUTE	116
HELP	117
MENU	118
SELECT	119
STOP	120
AGAIN	121
UNDO	122
CUT	123
COPY	124
PASTE	125
FIND	126
MUTE	127
VOLUMEUP	128
VOLUMEDOWN	129
KP_COMMA	133
KP_EQUALSAS400	134
INTERNATIONAL1	135
INTERNATIONAL2	136
INTERNATIONAL3	137
INTERNATIONAL4	138
INTERNATIONAL5	139
INTERNATIONAL6	140
INTERNATIONAL7	141
INTERNATIONAL8	142
INTERNATIONAL9	143
LANG1	144
LANG2	145
LANG3	146
LANG4	147
LANG5	148
LANG6	149
LANG7	150
LANG8	151
LANG9	152
ALTERASE	153
SYSREQ	154
CANCEL	155
CLEAR	156
PRIOR	157
RETURN2	158
SEPARATOR	159

continues on next page

Table 2 – continued from previous page

OUT	160
OPER	161
CLEARAGAIN	162
CRSEL	163
EXSEL	164
KP_00	176
KP_000	177
THOUSANDSSEPARATOR	178
DECIMALSEPARATOR	179
CURRENCYUNIT	180
CURRENCYSUBUNIT	181
KP_LEFTPAREN	182
KP_RIGHTPAREN	183
KP_LEFTBRACE	184
KP_RIGHTBRACE	185
KP_TAB	186
KP_BACKSPACE	187
KP_A	188
KP_B	189
KP_C	190
KP_D	191
KP_E	192
KP_F	193
KP_XOR	194
KP_POWER	195
KP_PERCENT	196
KP_LESS	197
KP_GREATER	198
KP_AMPERSAND	199
KP_DBLAMPERSAND	200
KP_VERTICALBAR	201
KP_DBLVERTICALBAR	202
KP_COLON	203
KP_HASH	204
KP_SPACE	205
KP_AT	206
KP_EXCLAM	207
KP_MEMSTORE	208
KP_MEMRECALL	209
KP_MEMCLEAR	210
KP_MEMADD	211
KP_MEMSUBTRACT	212
KP_MEMMULTIPLY	213
KP_MEMDIVIDE	214
KP_PLUSMINUS	215
KP_CLEAR	216
KP_CLEARENTRY	217
KP_BINARY	218
KP_OCTAL	219
KP_DECIMAL	220
KP_HEXADECIMAL	221
LCTRL	224

continues on next page

Table 2 – continued from previous page

LSHIFT	225
LALT	226
LGUI	227
RCTRL	228
RSHIFT	229
RALT	230
RGUI	231
MODE	257
AUDIONEXT	258
AUDIOPREV	259
AUDIOSTOP	260
AUDIOPLAY	261
AUDIOMUTE	262
MEDIASELECT	263
WWW	264
MAIL	265
CALCULATOR	266
COMPUTER	267
AC_SEARCH	268
AC_HOME	269
AC_BACK	270
AC_FORWARD	271
AC_STOP	272
AC_REFRESH	273
AC_BOOKMARKS	274
BRIGHTNESSDOWN	275
BRIGHTNESSUP	276
DISPLAYSWITCH	277
KBDILLUMTOGGLE	278
KBDILLUMDOWN	279
KBDILLUMUP	280
EJECT	281
SLEEP	282
APP1	283
APP2	284

__repr__() → str

Return the fully qualified name of this enum.

property keysym: *KeySym*

Return a *KeySym* from a scancode.

Based on the current keyboard layout.

property label: str

Return a human-readable name of a key based on its scancode.

Be sure not to confuse this with `.name`, which will return the enum name rather than the human-readable name.

See also:

KeySym.label

property scancode: *Scancode*

Return a scancode from a keycode.

Returns itself since it is already a *Scancode*.

See also:

KeySym.scancode

class tcod.event.**Modifier**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Keyboard modifier flags, a bit-field of held modifier keys.

Use *bitwise and* to check if a modifier key is held.

The following example shows some common ways of checking modifiers. All non-zero return values are considered true.

Example:

```
>>> mod = tcod.event.Modifier(4098)
>>> mod & tcod.event.Modifier.SHIFT  # Check if any shift key is held.
<Modifier.RSHIFT: 2>
>>> mod & tcod.event.Modifier.LSHIFT  # Check if left shift key is held.
<Modifier.NONE: 0>
>>> not mod & tcod.event.Modifier.LSHIFT  # Check if left shift key is NOT held.
True
>>> mod & tcod.event.Modifier.SHIFT and mod & tcod.event.Modifier.CTRL  # Check if␣
↪Shift+Control is held.
<Modifier.NONE: 0>
```

New in version 12.3.

NONE = 0

LSHIFT = 1

Left shift.

RSHIFT = 2

Right shift.

SHIFT = 3

LSHIFT | RSHIFT

LCTRL = 64

Left control.

RCTRL = 128

Right control.

CTRL = 192

LCTRL | RCTRL

LALT = 256

Left alt.

RALT = 512

Right alt.

ALT = 768
LALT | RALT

LGUI = 1024
Left meta key.

RGUI = 2048
Right meta key.

GUI = 3072
LGUI | RGUI

NUM = 4096
Numpad lock.

CAPS = 8192
Caps lock.

MODE = 16384
Alt graph.

CHAPTER

TWELVE

IMAGE HANDLING `TCOD.IMAGE`

Libtcod functionality for handling images.

This module is generally seen as outdated. To load images you should typically use Pillow or imageio unless you need to use a feature exclusive to libtcod.

Python-tcod is unable to render pixels to consoles. The best it can do with consoles is convert an image into semi-graphics which can be shown on non-emulated terminals. For true pixel-based rendering you'll want to access the SDL rendering port at *tcod.sdl.render*.

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

A libtcod image.

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:* 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: 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: 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: numpy.typing.ArrayLike*) → *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.

classmethod from_file(*path: str | PathLike[str]*) → *Image*

Return a new Image loaded from the given *path*.

New in version 16.0.

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: Console*) → None

Update an Image created with *libtcodpy.image_from_console*.

The console used with this function should have the same width and height as the Console given to *libtcodpy.image_from_console*. The font width and height must also be the same as when *libtcodpy.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 | PathLike[str]*) → None

Save the Image to a 32-bit .bmp or .png file.

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

Changed in version 16.0: Added PathLike support.

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 | PathLike[str]*) → NDArray[np.uint8]

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

THIRTEEN

LINE OF SIGHT `TCOD.LOS`

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

`tcod.los.`**bresenham**(*start: tuple[int, int]*, *end: tuple[int, int]*) → NDArray[np.intc]

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]))
>>> 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

FOURTEEN

FIELD OF VIEW TCOD.MAP

libtcod map attributes and field-of-view functions.

class tcod.map.**Map**(*width: int*, *height: int*, *order: Literal['C', '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],
```

(continues on next page)

(continued from previous page)

```
       [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: ArrayLike*, *pov: tuple[int, int]*, *radius: int = 0*, *light_walls: bool = True*, *algorithm: int = 12*) → NDArray[np.bool_]

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

FIFTEEN

NOISE MAP GENERATORS TCOD.NOISE

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.Algorithm.SIMPLEX,
...     seed=42,
... )
>>> samples = noise[tcod.noise.grid(shape=(5, 4), scale=0.25, origin=(0, 0))]
>>> samples  # Samples are a grid of floats between -1.0 and 1.0
array([[ 0.        , -0.55046356, -0.76072866, -0.7088647 , -0.68165785],
       [-0.27523372, -0.7205134 , -0.74057037, -0.43919194, -0.29195625],
       [-0.40398532, -0.57662135, -0.33160293,  0.12860827,  0.2864191 ],
       [-0.50773406, -0.2643614 ,  0.24446318,  0.6390255 ,  0.5922846 ]],
      dtype=float32)
>>> (samples + 1.0) * 0.5  # You can normalize samples to 0.0 - 1.0
array([[0.5       , 0.22476822, 0.11963567, 0.14556766, 0.15917107],
       [0.36238313, 0.1397433 , 0.12971482, 0.28040403, 0.35402188],
       [0.29800734, 0.21168932, 0.33419853, 0.5643041 , 0.6432096 ],
       [0.24613297, 0.3678193 , 0.6222316 , 0.8195127 , 0.79614234]],
      dtype=float32)
>>> ((samples + 1.0) * (256 / 2)).astype(np.uint8)  # Or as 8-bit unsigned bytes.
array([[128,  57,  30,  37,  40],
       [ 92,  35,  33,  71,  90],
       [ 76,  54,  85, 144, 164],
       [ 63,  94, 159, 209, 203]], dtype=uint8)
```

class tcod.noise.**Algorithm**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Libtcod noise algorithms.

New in version 12.2.

PERLIN = 1

Perlin noise.

SIMPLEX = 2

Simplex noise.

WAVELET = 4

Wavelet noise.

class tcod.noise.**Implementation**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Noise implementations.

New in version 12.2.

SIMPLE = 0

Generate plain noise.

FBM = 1

Fractional Brownian motion.

https://en.wikipedia.org/wiki/Fractional_Brownian_motion

TURBULENCE = 2

Turbulence noise implementation.

class tcod.noise.**Noise**(*dimensions: int*, *algorithm: int = Algorithm.SIMPLEX*, *implementation: int = Implementation.SIMPLE*, *hurst: float = 0.5*, *lacunarity: float = 2.0*, *octaves: float = 4*, *seed: int | Random | None = None*)

A configurable noise sampler.

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** – Must be from 1 to 4.
- **algorithm** – Defaults to *tcod.noise.Algorithm.SIMPLEX*
- **implementation** – Defaults to *tcod.noise.Implementation.SIMPLE*
- **hurst** – The hurst exponent. Should be in the 0.0-1.0 range.
- **lacunarity** – The noise lacunarity.
- **octaves** – The level of detail on fBm and turbulence implementations.
- **seed** – A Random instance, or None.

noise_c

A cffi pointer to a TCOD_noise_t object.

Type
CData

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** – The position on the 1st axis.
- **y** – The position on the 2nd axis.

- **z** – The position on the 3rd axis.
- **w** – The position on the 4th axis.

__getitem__(*indexes: Any*) → NDArray[np.float32]

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.

sample_mgrid(*mgrid: ArrayLike*) → NDArray[np.float32]

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 – 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*.

sample_ogrid(*ogrid: Sequence[ArrayLike]*) → NDArray[np.float32]

Sample an open mesh-grid array and return the result.

Parameters

ogrid – 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*.

tcod.noise.**grid**(*shape: tuple[int, ...]*, *scale: tuple[float, ...] | float*, *origin: tuple[int, ...] | None = None*, *indexing: Literal['ij', 'xy'] = 'xy'*) → tuple[NDArray[Any], ...]

Generate a mesh-grid of sample points to use with noise sampling.

Parameters

- **shape** – The shape of the grid. This can be any number of dimensions, but *Noise* classes only support up to 4.
- **scale** – The step size between samples. This can be a single float, or it can be a tuple of floats with one float for each axis in *shape*. A lower scale gives smoother transitions between noise values.
- **origin** – The position of the first sample. If *None* then the *origin* will be zero on each axis. *origin* is not scaled by the *scale* parameter.
- **indexing** – Passed to `numpy.meshgrid`.

Returns

A sparse mesh-grid to be passed into a *Noise* instance.

Example:

```
>>> noise = tcod.noise.Noise(dimensions=2, seed=42)

# Common case for ij-indexed arrays.
>>> noise[tcod.noise.grid(shape=(3, 5), scale=0.25, indexing="ij")]
array([[ 0.        , -0.27523372, -0.40398532, -0.50773406, -0.64945626],
       [-0.55046356, -0.7205134 , -0.57662135, -0.2643614 , -0.12529983],
       [-0.76072866, -0.74057037, -0.33160293,  0.24446318,  0.5346834 ]],
      dtype=float32)

# Transpose an xy-indexed array to get a standard order="F" result.
>>> noise[tcod.noise.grid(shape=(4, 5), scale=(0.5, 0.25), origin=(1.0, 1.0))].T
array([[ 0.52655405,  0.25038874, -0.03488023, -0.18455243, -0.16333057],
       [-0.5037453 , -0.75348294, -0.73630923, -0.35063767,  0.18149695],
       [-0.81221616, -0.6379566 , -0.12449139,  0.4495706 ,  0.7547447 ],
       [-0.7057655 , -0.5817767 , -0.22774395,  0.02399864, -0.07006818]],
      dtype=float32)
```

New in version 12.2.

CHAPTER

SIXTEEN

PATHFINDING TCOD.PATH

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*)

The older libtcod A* pathfinder.

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: NDArray[Any]*, *condition: ArrayLike | None = 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: ArrayLike*, *cost: NDArray[Any]*, *condition: ArrayLike | None = 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],
    [0, 1, 0],
]
```

(continues on next page)

(continued from previous page)

```
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],
    ],
    [
```

(continues on next page)

(continued from previous page)

```
        [1, 1, 1],
        [1, 0, 1],
        [1, 1, 1],
    ],
    [
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
    ],
]
```

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

Set 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.

property ndim: int

Return the number of dimensions.

property shape: tuple[int, ...]

Return the shape of this graph.

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

The older libtcod Dijkstra pathfinder.

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**(*array: numpy.typing.ArrayLike*)

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.typing.ArrayLike*) → *NodeCostArray*

Validate a numpy array and setup a C callback.

class tcod.path.**Pathfinder**(*graph:* CustomGraph | 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, ...]*) → NDArray[Any]

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:

```
>>> import tcod.path
>>> 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, ...]*) → NDArray[Any]

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:

```
>>> import tcod.path
>>> 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␣
```

(continues on next page)

(continued from previous page)

```
↪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: tuple[int, ...] | None = 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:

```
>>> import tcod.path
>>> 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]]...)
```

property distance: NDArray[Any]

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.

property traversal: NDArray[Any]

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.
>>> import tcod.path
>>> 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.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.typing.ArrayLike*, *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: ArrayLike*, *cost: ArrayLike*, *cardinal: int | None = None*, *diagonal: int | None = None*, *, *edge_map: ArrayLike | None = None*, *out: np.ndarray | None = Ellipsis*) → NDArray[Any]

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

distance is an input 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.

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.

out is the array to fill with the computed Dijkstra distance map. Having *out* be the same as *distance* will modify the array in-place, which is normally the fastest option. If *out* is *None* then the result is returned as a new array.

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)
```

(continues on next page)

(continued from previous page)

```
>>> dist[0, 0] = 0
>>> dist
array([[         0, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647],
       [2147483647, 2147483647, 2147483647]]...)
>>> tcod.path.dijkstra2d(dist, cost, 2, 3, out=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 = [
...     [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, out=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],
```

(continues on next page)

(continued from previous page)

```
       [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.

Changed in version 12.1: Added *out* parameter. Now returns the output array.

tcod.path.**hillclimb2d**(*distance: ArrayLike*, *start: tuple[int, int]*, *cardinal: bool | None = None*, *diagonal: bool | None = None*, *, *edge_map: ArrayLike | None = None*) → NDArray[Any]

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: DTypeLike = <class 'numpy.int32'>*, *order: Literal['C', 'F'] = 'C'*) → NDArray[Any]

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

SEVENTEEN

RANDOM NUMBER GENERATORS TCOD.RANDOM

Ports of the libtcod random number generator.

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: Hashable | None = 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

Warning: A non-integer seed is only deterministic if the environment variable `PYTHONHASHSEED` is set. In the future this function will only accept *int*'s as a seed.

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

__getstate__() → dict[str, Any]

Pack the self.random_c attribute into a portable state.

__setstate__(*state: dict[str, Any]*) → None

Create a new cdata object with the stored parameters.

gauss(*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

Changed in version 16.2: Renamed from *guass* to *gauss*.

inverse_gauss(*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

Changed in version 16.2: Renamed from *inverse_guass* to *inverse_gauss*.

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

EIGHTEEN

CONSOLE RENDERING EXTENSION TCOD.RENDER

Handles the rendering of libtcod's tilesets.

Using this module you can render a console to an SDL *Texture* directly, letting you have full control over how consoles are displayed. This includes rendering multiple tilesets in a single frame and rendering consoles on top of each other.

Example:

```
tileset = tcod.tileset.load_tilesheet("dejavu16x16_gs_tc.png", 32, 8, tcod.tileset.
↪CHARMAP_TCOD)
console = tcod.console.Console(20, 8)
console.print(0, 0, "Hello World")
sdl_window = tcod.sdl.video.new_window(
    console.width * tileset.tile_width,
    console.height * tileset.tile_height,
    flags=tcod.lib.SDL_WINDOW_RESIZABLE,
)
sdl_renderer = tcod.sdl.render.new_renderer(sdl_window, target_textures=True)
atlas = tcod.render.SDLTilesetAtlas(sdl_renderer, tileset)
console_render = tcod.render.SDLConsoleRender(atlas)
while True:
    sdl_renderer.copy(console_render.render(console))
    sdl_renderer.present()
    for event in tcod.event.wait():
        if isinstance(event, tcod.event.Quit):
            raise SystemExit()
```

New in version 13.4.

class tcod.render.**SDLConsoleRender**(*atlas:* SDLTilesetAtlas)

Holds an internal cache console and texture which are used to optimized console rendering.

render(*console:* Console) → *Texture*

Render a console to a cached Texture and then return the Texture.

You should not draw onto the returned Texture as only changed parts of it will be updated on the next call.

This function requires the SDL renderer to have target texture support. It will also change the SDL target texture for the duration of the call.

atlas: Final[*SDLTilesetAtlas*]

The SDLTilesetAtlas used to create this SDLConsoleRender.

New in version 13.7.

class tcod.render.**SDLTilesetAtlas**(*renderer:* Renderer, *tileset:* Tileset)

Prepares a tileset for rendering using SDL.

tileset: Final[*Tileset*]

The tileset used to create this SDLTilesetAtlas.

CHAPTER

NINETEEN

FONT LOADING FUNCTIONS TCOD.TILESET

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*) → NDArray[np.uint8]

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) → NDArray[np.uint8]

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: ArrayLike | NDArray[np.uint8]*) → None

Upload a tile into this array.

Parameters

- **codepoint** (*int*) – The Unicode codepoint you are assigning to. If the tile is a sprite rather than a common glyph then consider assigning it to a Private Use Area.
- **tile** (*Union[ArrayLike, NDArray[np.uint8]]*) – The pixels to use for this tile in row-major order and must be in the same shape as *tile_shape*. *tile* can be an RGBA array with the shape of `(height, width, rgba)`, or a grey-scale array with the shape `(height, width)`. The *tile* array will be converted to a dtype of `np.uint8`.

An RGB array as an input is too ambiguous and an alpha channel must be added, for example if an image has a key color than the key color pixels must have their alpha channel set to zero.

This data may be immediately sent to VRAM, which can be a slow operation.

Example:

```
# Examples use imageio for image loading, see https://imageio.readthedocs.io
tileset: tcod.tileset.Tileset  # This example assumes you are modifying an␣
↪existing tileset.

# Normal usage when a tile already has its own alpha channel.
# The loaded tile must be the correct shape for the tileset you assign it to.
# The tile is assigned to a private use area and will not conflict with any␣
↪exiting codepoint.
tileset.set_tile(0x100000, imageio.load("rgba_tile.png"))

# Load a greyscale tile.
tileset.set_tile(0x100001, imageio.load("greyscale_tile.png"), pilmode="L")
# If you are stuck with an RGB array then you can use the red channel as the␣
↪input: `rgb[:, :, 0]`

# Loads an RGB sprite without a background.
tileset.set_tile(0x100002, imageio.load("rgb_no_background.png", pilmode="RGBA
↪"))
# If you're stuck with an RGB array then you can pad the channel axis with an␣
↪alpha of 255:
#   rgba = np.pad(rgb, pad_width=((0, 0), (0, 0), (0, 1)), constant_values=255)

# Loads an RGB sprite with a key color background.
KEY_COLOR = np.asarray((255, 0, 255), dtype=np.uint8)
sprite_rgb = imageio.load("rgb_tile.png")
# Compare the RGB colors to KEY_COLOR, compress full matches to a 2D mask.
sprite_mask = (sprite_rgb != KEY_COLOR).all(axis=2)
# Generate the alpha array, with 255 as the foreground and 0 as the background.
sprite_alpha = sprite_mask.astype(np.uint8) * 255
# Combine the RGB and alpha arrays into an RGBA array.
sprite_rgba = np.append(sprite_rgb, sprite_alpha, axis=2)
tileset.set_tile(0x100003, sprite_rgba)
```

property tile_height: int

Height of the tile in pixels.

property tile_shape: tuple[int, int]

Shape (height, width) of the tile in pixels.

property tile_width: int

Width of the tile in pixels.

tcod.tileset.**get_default**() → *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 | PathLike[str]*) → *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 *libtcodpy.console_init_root* is called.

New in version 11.10.

tcod.tileset.**load_tilesheet**(*path: str | PathLike[str]*, *columns: int*, *rows: int*, *charmap: Iterable[int] | None*) → *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 a sequence of codepoints to map the tilesheet to in row-major order. 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 | PathLike[str]*, *tile_width: int*, *tile_height: int*) → *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.**procedural_block_elements**(*, *tileset:* Tileset) → None

Overwrite the block element codepoints in *tileset* with procedurally generated glyphs.

Parameters

tileset (Tileset) – A *Tileset* with tiles of any shape.

This will overwrite all of the codepoints listed here except for the shade glyphs.

This function is useful for other functions such as *Console.draw_semigraphics* which use more types of block elements than are found in Code Page 437.

New in version 13.1.

Example:

```
>>> tileset = tcod.tileset.Tileset(8, 8)
>>> tcod.tileset.procedural_block_elements(tileset=tileset)
>>> tileset.get_tile(0x259E)[:, :, 3]  # "" Quadrant upper right and lower left.
array([[  0,   0,   0,   0, 255, 255, 255, 255],
       [  0,   0,   0,   0, 255, 255, 255, 255],
       [  0,   0,   0,   0, 255, 255, 255, 255],
       [  0,   0,   0,   0, 255, 255, 255, 255],
       [255, 255, 255, 255,   0,   0,   0,   0],
       [255, 255, 255, 255,   0,   0,   0,   0],
       [255, 255, 255, 255,   0,   0,   0,   0],
       [255, 255, 255, 255,   0,   0,   0,   0]], dtype=uint8)
>>> tileset.get_tile(0x2581)[:, :, 3]  # "" Lower one eighth block.
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,   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,   0,   0,   0,   0,   0,   0],
       [255, 255, 255, 255, 255, 255, 255, 255]], dtype=uint8)
>>> tileset.get_tile(0x258D)[:, :, 3]  # "" Left three eighths block.
array([[255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0],
       [255, 255, 255,   0,   0,   0,   0,   0]], dtype=uint8)
```

tcod.tileset.**set_default**(*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 | PathLike[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 *libtcodpy.console_init_root*. Once the root console is setup you may call this function 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, 9689, 9794, 9792, 9834, 9835, 9788, 9658, 9668, 8597, 8252, 182, 167, 9644, 8616, 8593, 8595, 8594, 8592, 8735, 8596, 9650, 9660, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 8962, 199, 252, 233, 226, 228, 224, 229, 231, 234, 235, 232, 239, 238, 236, 196, 197, 201, 230, 198, 244, 246, 242, 251, 249, 255, 214, 220, 162, 163, 165, 8359, 402, 225, 237, 243, 250, 241, 209, 170, 186, 191, 8976, 172, 189, 188, 161, 171, 187, 9617, 9618, 9619, 9474, 9508, 9569, 9570, 9558, 9557, 9571, 9553, 9559, 9565, 9564, 9563, 9488, 9492, 9524, 9516, 9500, 9472, 9532, 9566, 9567, 9562, 9556, 9577, 9574, 9568, 9552, 9580, 9575, 9576, 9572, 9573, 9561, 9560, 9554, 9555, 9579, 9578, 9496, 9484, 9608, 9604, 9612, 9616, 9600, 945, 223, 915, 960, 931, 963, 181, 964, 934, 920, 937, 948, 8734, 966, 949, 8745, 8801, 177, 8805, 8804, 8992, 8993, 247, 8776, 176, 8729, 183, 8730, 8319, 178, 9632, 160]**

A code page 437 character mapping.

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

New in version 11.12.

Changed in version 14.0: Character at index `0x7F` was changed from value `0x7F` to the HOUSE glyph `0x2302`.

tcod.tileset.**CHARMAP_TCOD = [32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 91, 92, 93, 94, 95, 96, 123, 124, 125, 126, 9617, 9618, 9619, 9474, 9472, 9532, 9508, 9524, 9500, 9516, 9492, 9484, 9488, 9496, 9624, 9629, 9600, 9622, 9626, 9616, 9623, 8593, 8595, 8592, 8594, 9650, 9660, 9668, 9658, 8597, 8596, 9744, 9745, 9675, 9673, 9553, 9552, 9580, 9571, 9577, 9568, 9574, 9562, 9556, 9559, 9565, 0, 0, 0, 0, 0, 0, 0, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 0, 0, 0, 0, 0, 0, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 0, 0, 0, 0, 0, 0]**

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

TWENTY

OLD API FUNCTIONS `LIBTCODPY`

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.

Use `from tcod import libtcodpy` to access this module.

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.

20.1 bsp

libtcodpy.**bsp_new_with_size**(*x: int*, *y: int*, *w: int*, *h: int*) → *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.

libtcodpy.**bsp_split_once**(*node: BSP*, *horizontal: bool*, *position: int*) → None

Deprecated function.

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

libtcodpy.**bsp_split_recursive**(*node: BSP*, *randomizer: Random | None*, *nb: int*, *minHSize: int*, *minVSize: int*, *maxHRatio: float*, *maxVRatio: float*) → None

Deprecated function.

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

libtcodpy.**bsp_resize**(*node:* BSP, *x:* *int*, *y:* *int*, *w:* *int*, *h:* *int*) → None

Deprecated function.

Deprecated since version 2.0: Assign directly to *BSP* attributes instead.

libtcodpy.**bsp_left**(*node:* BSP) → *BSP* | None

Deprecated function.

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

libtcodpy.**bsp_right**(*node:* BSP) → *BSP* | None

Deprecated function.

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

libtcodpy.**bsp_father**(*node:* BSP) → *BSP* | None

Deprecated function.

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

libtcodpy.**bsp_is_leaf**(*node:* BSP) → bool

Deprecated function.

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

libtcodpy.**bsp_contains**(*node:* BSP, *cx:* *int*, *cy:* *int*) → bool

Deprecated function.

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

libtcodpy.**bsp_find_node**(*node:* BSP, *cx:* *int*, *cy:* *int*) → *BSP* | None

Deprecated function.

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

libtcodpy.**bsp_traverse_pre_order**(*node:* BSP, *callback:* *Callable[[*BSP, *Any], None]*, *userData:* *Any* = *0*) → None

Traverse this nodes hierarchy with a callback.

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

libtcodpy.**bsp_traverse_in_order**(*node:* BSP, *callback:* *Callable[[*BSP, *Any], None]*, *userData:* *Any* = *0*) → None

Traverse this nodes hierarchy with a callback.

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

libtcodpy.**bsp_traverse_post_order**(*node:* BSP, *callback:* *Callable[[*BSP, *Any], None]*, *userData:* *Any* = *0*) → None

Traverse this nodes hierarchy with a callback.

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

libtcodpy.**bsp_traverse_level_order**(*node:* BSP, *callback:* *Callable[[*BSP, *Any], None]*, *userData:* *Any* = *0*) → None

Traverse this nodes hierarchy with a callback.

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

libtcodpy.**bsp_traverse_inverted_level_order**(*node:* BSP, *callback:* *Callable[[*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.

libtcodpy.**bsp_remove_sons**(*node:* 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.

libtcodpy.**bsp_delete**(*node:* 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.

20.2 color

class libtcodpy.**Color**(*r:* *int* = *0*, *g:* *int* = *0*, *b:* *int* = *0*)

Old-style libtcodpy color class.

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.

property r: int

Red value, always normalized to 0-255.

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

Type
int

property g: int

Green value, always normalized to 0-255.

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

Type
int

property b: int

Blue value, always normalized to 0-255.

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

Type
int

__getitem__(*index: Any*) → Any

Return a color channel.

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

__eq__(*other: object*) → bool

Compare equality between colors.

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

__add__(*other: object*) → *Color*

Add two colors together.

Deprecated since version 9.2: Use NumPy instead for color math operations.

__sub__(*other: object*) → *Color*

Subtract one color from another.

Deprecated since version 9.2: Use NumPy instead for color math operations.

__mul__(*other: object*) → *Color*

Multiply with a scaler or another color.

Deprecated since version 9.2: Use NumPy instead for color math operations.

__repr__() → str

Return a printable representation of the current color.

libtcodpy.**color_lerp**(*c1: tuple[int, int, int]*, *c2: tuple[int, int, int]*, *a: float*) → *Color*

Return the linear interpolation between two colors.

`a` is the interpolation value, with 0 returning `c1`, 1 returning `c2`, and 0.5 returning 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

libtcodpy.**color_set_hsv**(*c:* 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 in-place.

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.

libtcodpy.**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]

libtcodpy.**color_scale_HSV**(*c:* Color, *scoef: float*, *vcoef: float*) → None

Scale a color's saturation and value.

Does not return a new Color. `c` is modified in-place.

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.

libtcodpy.**color_gen_map**(*colors: Iterable[tuple[int, int, int]]*, *indexes: Iterable[int]*) → list[*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)]
```

20.2.1 color controls

Libtcod color control constants. These can be inserted into Python strings with the %c format specifier as shown below.

libtcodpy.**COLCTRL_1**

These can be configured with *libtcodpy.console_set_color_control*. However, it is recommended to use *libtcodpy.COLCTRL_FORE_RGB* and *libtcodpy.COLCTRL_BACK_RGB* instead.

libtcodpy.**COLCTRL_2**

libtcodpy.**COLCTRL_3**

libtcodpy.**COLCTRL_4**

libtcodpy.**COLCTRL_5**

libtcodpy.**COLCTRL_STOP**

When this control character is inserted into a string the foreground and background colors will be reset for the remaining characters of the string.

```
>>> import tcod
>>> reset_color = f"{libtcodpy.COLCTRL_STOP:c}"
```

libtcodpy.**COLCTRL_FORE_RGB**

Sets the foreground color to the next 3 Unicode characters for the remaining characters.

```
>>> fg = (255, 255, 255)
>>> change_fg = f"{libtcodpy.COLCTRL_FORE_RGB:c}{fg[0]:c}{fg[1]:c}{fg[2]:c}"
>>> string = f"Old color {change_fg}new color{libtcodpy.COLCTRL_STOP:c} old color."
```

libtcodpy.**COLCTRL_BACK_RGB**

Sets the background color to the next 3 Unicode characters for the remaining characters.

```
>>> from typing import Tuple
>>> def change_colors(fg: Tuple[int, int, int], bg: Tuple[int, int, int]) -> str:
...     """Return the control codes to change the foreground and background colors."
↪""
...     return "%c%c%c%c%c%c%c%c" % (libtcodpy.COLCTRL_FORE_RGB, *fg, libtcodpy.
↪COLCTRL_BACK_RGB, *bg)
>>> string = f"Old {change_colors(fg=(255, 255, 255), bg=(0, 0, 255))}new"
```

20.3 console

libtcodpy.**console_set_custom_font**(*fontFile: str | PathLike[str]*, *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 *libtcodpy.console_init_root*.

Flags can be a mix of the following:

- libtcodpy.FONT_LAYOUT_ASCII_INCOL: Decode tileset raw in column-major order.
- libtcodpy.FONT_LAYOUT_ASCII_INROW: Decode tileset raw in row-major order.

- libtcodpy.FONT_TYPE_GREYSCALE: Force tileset to be read as greyscale.
- libtcodpy.FONT_TYPE_GRAYSCALE
- libtcodpy.FONT_LAYOUT_TCOD: Unique layout used by libtcod.
- libtcodpy.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_tilesheet` instead. See *Getting Started* for more info.

Changed in version 16.0: Added PathLike support. *fontFile* no longer takes bytes.

libtcodpy.**console_init_root**(*w: int*, *h: int*, *title: str | None = None*, *fullscreen: bool = False*, *renderer: int | None = None*, *order: Literal['C', 'F'] = 'C'*, *vsync: bool | None = None*) → *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 libtcodpy.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.

libtcodpy.**console_flush**(*console: Console | None = None*, ***, *keep_aspect: bool = False*, *integer_scaling: bool = False*, *snap_to_integer: bool | None = None*, *clear_color: 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:

libtcodpy.console_init_root `tcod.console.recommended_size`

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

libtcodpy.**console_blit**(*src: Console*, *x: int*, *y: int*, *w: int*, *h: int*, *dst: 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.

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

Return a recently pressed key.

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

Example:

```
for event in tcod.event.get():
    if isinstance(event, tcod.event.KeyDown):
        ...
```

libtcodpy.**console_clear**(*con: 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.

libtcodpy.**console_credits**() → None

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

libtcodpy.**console_credits_reset**() → None

libtcodpy.**console_delete**(*con:* 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.

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

Fill the background 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.

libtcodpy.**console_fill_char**(*con:* 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.

libtcodpy.**console_fill_foreground**(*con:* Console, *r: Sequence[int]*, *g: Sequence[int]*, *b: Sequence[int]*) → None

Fill the foreground 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.

libtcodpy.**console_from_file**(*filename: str | PathLike[str]*) → *Console*

Return a new console object from a filename.

The file format is automatically 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.

Deprecated since version 12.7: Use *libtcodpy.console_load_xp* to load REXPaint consoles.

Other formats are not actively supported.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_from_xp**(*filename: str | PathLike[str]*) → *Console*

Return a single console from a REXPaint *.xp* file.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_get_alignment**(*con:* Console) → int

Return this consoles current alignment mode.

Parameters
con (Console) – Any Console instance.

Deprecated since version 8.5: Check *Console.default_alignment* instead.

libtcodpy.**console_get_background_flag**(*con:* 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.

libtcodpy.**console_get_char**(*con:* 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*.

libtcodpy.**console_get_char_background**(*con:* Console, *x: int*, *y: int*) → *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*.

libtcodpy.**console_get_char_foreground**(*con:* Console, *x: int*, *y: int*) → *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*.

libtcodpy.**console_get_default_background**(*con:* Console) → *Color*

Return this consoles default background color.

Deprecated since version 8.5: Use *Console.default_bg* instead.

libtcodpy.**console_get_default_foreground**(*con:* Console) → *Color*

Return this consoles default foreground color.

Deprecated since version 8.5: Use *Console.default_fg* instead.

libtcodpy.**console_get_fade**() → int

Deprecated function.

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

libtcodpy.**console_get_fading_color**() → *Color*

Deprecated function.

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

libtcodpy.**console_get_height**(*con:* 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.

libtcodpy.**console_get_height_rect**(*con:* 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.

libtcodpy.**console_get_width**(*con:* 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.

libtcodpy.**console_hline**(*con:* 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.

libtcodpy.**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.

libtcodpy.**console_is_key_pressed**(*key: int*) → bool

Return True if a key is held.

Deprecated since version 12.7: Use `tcod.event.get_keyboard_state` to check if a key is held.

libtcodpy.**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.

libtcodpy.**console_load_apf**(*con: Console*, *filename: str | PathLike[str]*) → bool

Update a console from an ASCII Paint *.apf* file.

Deprecated since version 12.7: This format is no longer supported.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_load_asc**(*con: Console*, *filename: str | PathLike[str]*) → bool

Update a console from a non-delimited ASCII *.asc* file.

Deprecated since version 12.7: This format is no longer supported.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_load_xp**(*con: Console*, *filename: str | PathLike[str]*) → bool

Update a console from a REXPaint *.xp* file.

Deprecated since version 11.18: Functions modifying console objects in-place are deprecated. Use `libtcodpy.console_from_xp` to load a Console from a file.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_list_load_xp**(*filename: str | PathLike[str]*) → list[*Console*] | None

Return a list of consoles from a REXPaint *.xp* file.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_list_save_xp**(*console_list: Sequence[Console]*, *filename: str | PathLike[str]*, *compress_level: int = 9*) → bool

Save a list of consoles to a REXPaint *.xp* file.

Changed in version 16.0: Added PathLike support.

libtcodpy.**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.

libtcodpy.**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.

libtcodpy.**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. Any null character *'x00'* will prematurely end the printed text.
- **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.

libtcodpy.**console_new**(*w: int*, *h: int*) → *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.

libtcodpy.**console_print**(*con:* 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 optionally using color codes.

Deprecated since version 8.5: Use *Console.print_* instead.

libtcodpy.**console_print_ex**(*con:* 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.
- **flag** – Blending mode to use.
- **alignment** – The libtcod alignment constant.
- **fmt** – A unicode or bytes string, optionally using color codes.

Deprecated since version 8.5: Use *Console.print_* instead.

libtcodpy.**console_print_frame**(*con: Console*, *x: int*, *y: int*, *w: int*, *h: int*, *clear: bool = True*, *flag: int = 13*, *fmt: str = ''*) → 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.

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.

libtcodpy.**console_print_rect**(*con: 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.

libtcodpy.**console_print_rect_ex**(*con: 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.

libtcodpy.**console_put_char**(*con: Console*, *x: int*, *y: int*, *c: 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.

libtcodpy.**console_put_char_ex**(*con: Console*, *x: int*, *y: int*, *c: 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.

libtcodpy.**console_rect**(*con:* 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.

libtcodpy.**console_save_apf**(*con:* Console, *filename:* *str* | *PathLike[str]*) → bool

Save a console to an ASCII Paint *.apf* file.

Deprecated since version 12.7: This format is no longer supported.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_save_asc**(*con:* Console, *filename:* *str* | *PathLike[str]*) → bool

Save a console to a non-delimited ASCII *.asc* file.

Deprecated since version 12.7: This format is no longer supported.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_save_xp**(*con:* Console, *filename:* *str* | *PathLike[str]*, *compress_level:* *int* = *9*) → bool

Save a console to a REXPaint *.xp* file.

Changed in version 16.0: Added PathLike support.

libtcodpy.**console_set_alignment**(*con:* Console, *alignment:* *int*) → None

Change this consoles current alignment mode.

- tcod.LEFT
- tcod.CENTER
- tcod.RIGHT

Parameters

- **con** (`Console`) – Any Console instance.
- **alignment** (*int*) – The libtcod alignment constant.

Deprecated since version 8.5: Set `Console.default_alignment` instead.

libtcodpy.**console_set_background_flag**(*con:* 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.

libtcodpy.**console_set_char**(*con:* Console, *x:* *int*, *y:* *int*, *c:* *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*.

libtcodpy.**console_set_char_background**(*con:* 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.

libtcodpy.**console_set_char_foreground**(*con:* 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*.

libtcodpy.**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.

libtcodpy.**console_set_default_background**(*con:* 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.

libtcodpy.**console_set_default_foreground**(*con:* 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.

libtcodpy.**console_set_fade**(*fade:* *int*, *fadingColor:* *tuple[int, int, int]*) → None

Deprecated function.

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

libtcodpy.**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.

libtcodpy.**console_set_key_color**(*con:* 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.

libtcodpy.**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.

libtcodpy.**console_vline**(*con:* 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.

libtcodpy.**console_wait_for_keypress**(*flush:* *bool*) → *Key*

Block until the user presses a key, then returns a new Key.

Parameters

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.

Example:

```
for event in tcod.event.wait():
    if isinstance(event, tcod.event.KeyDown):
        ...
```

20.4 Event

class libtcodpy.**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] == '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 libtcodpy.**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.

20.4.1 Event Types

libtcodpy.**EVENT_NONE**

libtcodpy.**EVENT_KEY_PRESS**

libtcodpy.**EVENT_KEY_RELEASE**

libtcodpy.**EVENT_KEY**

Same as libtcodpy.EVENT_KEY_PRESS | libtcodpy.EVENT_KEY_RELEASE

libtcodpy.**EVENT_MOUSE_MOVE**

libtcodpy.**EVENT_MOUSE_PRESS**

libtcodpy.**EVENT_MOUSE_RELEASE**

libtcodpy.**EVENT_MOUSE**

Same as libtcodpy.EVENT_MOUSE_MOVE | libtcodpy.EVENT_MOUSE_PRESS | libtcodpy.EVENT_MOUSE_RELEASE

libtcodpy.**EVENT_FINGER_MOVE**

libtcodpy.**EVENT_FINGER_PRESS**

libtcodpy.**EVENT_FINGER_RELEASE**

libtcodpy.**EVENT_FINGER**

Same as libtcodpy.EVENT_FINGER_MOVE | libtcodpy.EVENT_FINGER_PRESS | libtcodpy.EVENT_FINGER_RELEASE

libtcodpy.**EVENT_ANY**

Same as libtcodpy.EVENT_KEY | libtcodpy.EVENT_MOUSE | libtcodpy.EVENT_FINGER

20.5 sys

libtcodpy.**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.

libtcodpy.**sys_get_fps**() → int

Return the current frames per second.

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

libtcodpy.**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.

libtcodpy.**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.

libtcodpy.**sys_elapsed_milli**() → int

Get number of milliseconds since the start of the program.

Returns
Time since the program has started in milliseconds.

Return type
int

Deprecated since version 2.0: Use Python's `time` module instead.

libtcodpy.**sys_elapsed_seconds**() → float

Get number of seconds since the start of the program.

Returns
Time since the program has started in seconds.

Return type
float

Deprecated since version 2.0: Use Python's `time` module instead.

libtcodpy.**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.

libtcodpy.**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.

libtcodpy.**sys_save_screenshot**(*name: str | PathLike[str] | None = 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
name – File path to save screenshot.

Deprecated since version 11.13: This function is not supported by contexts. Use *Context.save_screenshot* instead.

Changed in version 16.0: Added PathLike support.

libtcodpy.**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.

libtcodpy.**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.

libtcodpy.**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.

libtcodpy.**sys_update_char**(*asciiCode: int*, *fontx: int*, *fonty: int*, *img: 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 *libtcodpy.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.

libtcodpy.**sys_register_SDL_renderer**(*callback: Callable[[Any], None]*) → None

Register a custom rendering function with libtcod.

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

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

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

Parameters

callback – A function which takes a single argument.

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

libtcodpy.**sys_check_for_event**(*mask: int*, *k:* Key | *None*, *m:* Mouse | *None*) → 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.

libtcodpy.**sys_wait_for_event**(*mask: int*, *k:* Key | *None*, *m:* Mouse | *None*, *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 received.

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.

20.6 pathfinding

libtcodpy.**dijkstra_compute**(*p:* Dijkstra, *ox:* *int*, *oy:* *int*) → None

libtcodpy.**dijkstra_delete**(*p:* Dijkstra) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**dijkstra_get**(*p:* Dijkstra, *idx:* *int*) → tuple[int, int]

libtcodpy.**dijkstra_get_distance**(*p:* Dijkstra, *x:* *int*, *y:* *int*) → int

libtcodpy.**dijkstra_is_empty**(*p:* Dijkstra) → bool

libtcodpy.**dijkstra_new**(*m:* Map, *dcost:* *float* = *1.41*) → *Dijkstra*

libtcodpy.**dijkstra_new_using_function**(*w:* *int*, *h:* *int*, *func:* *Callable[[int, int, int, int, Any], float]*, *userData:* *Any* = *0*, *dcost:* *float* = *1.41*) → *Dijkstra*

libtcodpy.**dijkstra_path_set**(*p:* Dijkstra, *x:* *int*, *y:* *int*) → bool

libtcodpy.**dijkstra_path_walk**(*p:* Dijkstra) → tuple[int, int] | tuple[None, None]

libtcodpy.**dijkstra_reverse**(*p:* Dijkstra) → None

libtcodpy.**dijkstra_size**(*p:* Dijkstra) → int

libtcodpy.**path_compute**(*p:* 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

libtcodpy.**path_delete**(*p:* AStar) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**path_get**(*p:* 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*

libtcodpy.**path_get_destination**(*p:* 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]

libtcodpy.**path_get_origin**(*p:* 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]

libtcodpy.**path_is_empty**(*p:* 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

libtcodpy.**path_new_using_function**(*w: int*, *h: int*, *func: Callable[[int, int, int, int, Any], float]*, *userData: Any = 0*, *dcost: float = 1.41*) → *AStar*

Return a new AStar using the given callable function.

Parameters

- **w** (*int*) – Clipping width.
- **h** (*int*) – Clipping height.
- **func** – Callback function with the format: *f(origin_x, origin_y, dest_x, dest_y, userData) -> float*
- **userData** (*Any*) – An object passed to the callback.
- **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

libtcodpy.**path_new_using_map**(*m:* Map, *dcost: float = 1.41*) → *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

libtcodpy.**path_reverse**(*p:* AStar) → None

Reverse the direction of a path.

This effectively swaps the origin and destination points.

Parameters
p (AStar) – An AStar instance.

libtcodpy.**path_size**(*p:* AStar) → int

Return the current length of the computed path.

Parameters
p (AStar) – An AStar instance.

Returns
Length of the path.

Return type
int

libtcodpy.**path_walk**(*p:* AStar, *recompute: bool*) → 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]]

20.7 heightmap

libtcodpy.**heightmap_add**(*hm: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_add_fbm**(*hm: NDArray[np.float32]*, *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*.

libtcodpy.**heightmap_add_hill**(*hm: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_add_hm**(*hm1: NDArray[np.float32]*, *hm2: NDArray[np.float32]*, *hm3: NDArray[np.float32]*) → 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.

libtcodpy.**heightmap_add_voronoi**(*hm: NDArray[np.float32]*, *nbPoints: Any*, *nbCoef: int*, *coef: Sequence[float]*, *rnd:* tcod.random.Random | *None* = *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.

libtcodpy.**heightmap_clamp**(*hm: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_clear**(*hm: NDArray[np.float32]*) → 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.

libtcodpy.**heightmap_copy**(*hm1: NDArray[np.float32]*, *hm2: NDArray[np.float32]*) → None

Copy the heightmap `hm1` to `hm2`.

Parameters

- **hm** – A numpy.ndarray formatted for heightmap functions.
- **hm1** (*numpy.ndarray*) – The source heightmap.
- **hm2** (*numpy.ndarray*) – The destination heightmap.

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

libtcodpy.**heightmap_count_cells**(*hm: NDArray[np.float32]*, *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))`

libtcodpy.**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.

libtcodpy.**heightmap_dig_bezier**(*hm: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_dig_hill**(*hm: NDArray[np.float32]*, *x: float*, *y: float*, *radius: float*, *height: float*) → None

Dig a hill in a heightmap.

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.

libtcodpy.**heightmap_get_interpolated_value**(*hm: NDArray[np.float32]*, *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

libtcodpy.**heightmap_get_minmax**(*hm: NDArray[np.float32]*) → 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.

libtcodpy.**heightmap_get_normal**(*hm: NDArray[np.float32]*, *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]

libtcodpy.**heightmap_get_slope**(*hm: NDArray[np.float32]*, *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

libtcodpy.**heightmap_get_value**(*hm: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_has_land_on_border**(*hm: NDArray[np.float32]*, *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

libtcodpy.**heightmap_kernel_transform**(*hm: NDArray[np.float32]*, *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 neighbor 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 coordinates.
- **dy** (*Sequence[int]*) – A sequence of y coordinates.
- **weight** (*Sequence[float]*) – A sequence of kernelSize cells weight. The value of each neighbor 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)
```

libtcodpy.**heightmap_lerp_hm**(*hm1: NDArray[np.float32]*, *hm2: NDArray[np.float32]*, *hm3: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_multiply_hm**(*hm1: NDArray[np.float32]*, *hm2: NDArray[np.float32]*, *hm3: NDArray[np.float32]*) → 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[:])`.

libtcodpy.**heightmap_new**(*w: int*, *h: int*, *order: str = 'C'*) → NDArray[np.float32]

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.

libtcodpy.**heightmap_normalize**(*hm: NDArray[np.float32]*, *mi: float = 0.0*, *ma: float = 1.0*) → None

Normalize heightmap values between `mi` and `ma`.

Parameters

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

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

libtcodpy.**heightmap_rain_erosion**(*hm: NDArray[np.float32]*, *nbDrops: int*, *erosionCoef: float*, *sedimentationCoef: float*, *rnd:* tcod.random.Random | *None* = *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.

libtcodpy.**heightmap_scale**(*hm: NDArray[np.float32]*, *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.

libtcodpy.**heightmap_scale_fbm**(*hm: NDArray[np.float32]*, *noise:* tcod.noise.Noise, *mulx: float*, *muly: float*, *addx: float*, *addy: float*, *octaves: float*, *delta: float*, *scale: float*) → None

Multiply the heightmap 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*.

libtcodpy.**heightmap_set_value**(*hm: NDArray[np.float32]*, *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.

20.8 image

libtcodpy.**image_load**(*filename: str | PathLike[str]*) → *Image*

Load an image file into an Image instance and return it.

Parameters
filename – Path to a .bmp or .png image file.

Changed in version 16.0: Added PathLike support.

Deprecated since version 16.0: Use *tcod.image.Image.from_file* instead.

libtcodpy.**image_from_console**(*console:* Console) → *Image*

Return an Image with a Consoles pixel data.

This effectively takes a screen-shot of the Console.

Parameters
console (Console) – Any Console instance.

Deprecated since version 16.0: *Tileset.render* is a better alternative.

libtcodpy.**image_blit**(*image:* Image, *console:* Console, *x: float*, *y: float*, *bkgnd_flag: int*, *scalex: float*, *scaley: float*, *angle: float*) → None

libtcodpy.**image_blit_2x**(*image:* Image, *console:* Console, *dx: int*, *dy: int*, *sx: int = 0*, *sy: int = 0*, *w: int = -1*, *h: int = -1*) → None

libtcodpy.**image_blit_rect**(*image:* Image, *console:* Console, *x: int*, *y: int*, *w: int*, *h: int*, *bkgnd_flag: int*) → None

libtcodpy.**image_clear**(*image:* Image, *col: tuple[int, int, int]*) → None

libtcodpy.**image_delete**(*image:* Image) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**image_get_alpha**(*image:* Image, *x: int*, *y: int*) → int

libtcodpy.**image_get_mipmap_pixel**(*image:* Image, *x0: float*, *y0: float*, *x1: float*, *y1: float*) → tuple[int, int, int]

libtcodpy.**image_get_pixel**(*image:* Image, *x: int*, *y: int*) → tuple[int, int, int]

libtcodpy.**image_get_size**(*image:* Image) → tuple[int, int]

libtcodpy.**image_hflip**(*image:* Image) → None

libtcodpy.**image_invert**(*image:* Image) → None

libtcodpy.**image_is_pixel_transparent**(*image:* Image, *x: int*, *y: int*) → bool

libtcodpy.**image_new**(*width: int*, *height: int*) → *Image*

libtcodpy.**image_put_pixel**(*image:* Image, *x: int*, *y: int*, *col: tuple[int, int, int]*) → None

libtcodpy.**image_refresh_console**(*image:* Image, *console:* Console) → None

Update an image made with *image_from_console*.

Deprecated since version 16.0: This function is unnecessary, use *Tileset.render* instead.

libtcodpy.**image_rotate90**(*image:* Image, *num:* *int* = *1*) → None

libtcodpy.**image_save**(*image:* Image, *filename:* *str* | *PathLike[str]*) → None

libtcodpy.**image_scale**(*image:* Image, *neww:* *int*, *newh:* *int*) → None

libtcodpy.**image_set_key_color**(*image:* Image, *col:* *tuple[int, int, int]*) → None

libtcodpy.**image_vflip**(*image:* Image) → None

20.9 line

libtcodpy.**line_init**(*xo:* *int*, *yo:* *int*, *xd:* *int*, *yd:* *int*) → None

Initialize 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: This function was replaced by *tcod.los.bresenham*.

libtcodpy.**line_step**() → 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: This function was replaced by *tcod.los.bresenham*.

libtcodpy.**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 interaction by returning False or None, otherwise True.

Return type
bool

Deprecated since version 2.0: This function was replaced by `tcod.los.bresenham`.

libtcodpy.**line_iter**(*xo: int*, *yo: int*, *xd: int*, *yd: int*) → Iterator[tuple[int, int]]

Returns an Iterable over a Bresenham line.

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`.

libtcodpy.**line_where**(*x1: int*, *y1: int*, *x2: int*, *y2: int*, *inclusive: bool = True*) → tuple[NDArray[np.intc], NDArray[np.intc]]

Return a NumPy index array following a Bresenham line.

If *inclusive* is true then the start point is included in the result.

New in version 4.6.

Deprecated since version 11.14: This function was replaced by `tcod.los.bresenham`.

20.10 map

libtcodpy.**map_clear**(*m: 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.

libtcodpy.**map_compute_fov**(*m: 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.

libtcodpy.**map_copy**(*source:* Map, *dest:* 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.

libtcodpy.**map_delete**(*m:* Map) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**map_get_height**(*map:* Map) → int

Return the height of a map.

Deprecated since version 4.5: Check the `tcod.map.Map.height` attribute instead.

libtcodpy.**map_get_width**(*map:* Map) → int

Return the width of a map.

Deprecated since version 4.5: Check the `tcod.map.Map.width` attribute instead.

libtcodpy.**map_is_in_fov**(*m:* 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.

libtcodpy.**map_is_transparent**(*m:* Map, *x:* *int*, *y:* *int*) → bool

Return True is a map cell is transparent.

Note: This function is slow.

Deprecated since version 4.5: Use `tcod.map.Map.transparent` to check this property.

libtcodpy.**map_is_walkable**(*m:* Map, *x:* *int*, *y:* *int*) → bool

Return True is a map cell is walkable.

Note: This function is slow.

Deprecated since version 4.5: Use `tcod.map.Map.walkable` to check this property.

libtcodpy.**map_new**(*w:* *int*, *h:* *int*) → *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.

libtcodpy.**map_set_properties**(*m:* 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.

20.11 mouse

libtcodpy.**mouse_get_status**() → *Mouse*

libtcodpy.**mouse_is_cursor_visible**() → bool

Return True if the mouse cursor is visible.

Deprecated since version 16.0: Use `tcod.sdl.mouse.show` instead.

libtcodpy.**mouse_move**(*x: int*, *y: int*) → None

libtcodpy.**mouse_show_cursor**(*visible: bool*) → None

Change the visibility of the mouse cursor.

Deprecated since version 16.0: Use `tcod.sdl.mouse.show` instead.

20.12 namegen

libtcodpy.**namegen_destroy**() → None

libtcodpy.**namegen_generate**(*name: str*) → str

libtcodpy.**namegen_generate_custom**(*name: str*, *rule: str*) → str

libtcodpy.**namegen_get_sets**() → list[str]

libtcodpy.**namegen_parse**(*filename: str | PathLike[str]*, *random: Random | None = None*) → None

20.13 noise

libtcodpy.**noise_delete**(*n:* Noise) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**noise_get**(*n:* 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 the *dimensions* attribute of `Noise`. If `f` is shorter than *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

libtcodpy.**noise_get_fbm**(*n:* 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.
- **oc** (*float*) – The level of level. Should be more than 1.

Returns
The sampled noise value.

Return type
float

libtcodpy.**noise_get_turbulence**(*n:* 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.
- **oc** (*float*) – The level of level. Should be more than 1.

Returns
The sampled noise value.

Return type
float

libtcodpy.**noise_new**(*dim: int*, *h: float = 0.5*, *l: float = 2.0*, *random:* Random | *None = None*) → *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

libtcodpy.**noise_set_type**(*n:* Noise, *typ: int*) → None

Set a Noise objects default noise algorithm.

Parameters

- **n** – Noise object.
- **typ** (*int*) – Any NOISE_* constant.

20.14 parser

libtcodpy.**parser_delete**(*parser: Any*) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**parser_get_bool_property**(*parser: Any*, *name: str*) → bool

libtcodpy.**parser_get_char_property**(*parser: Any*, *name: str*) → str

libtcodpy.**parser_get_color_property**(*parser: Any*, *name: str*) → *Color*

libtcodpy.**parser_get_dice_property**(*parser: Any*, *name: str*) → *Dice*

libtcodpy.**parser_get_float_property**(*parser: Any*, *name: str*) → float

libtcodpy.**parser_get_int_property**(*parser: Any*, *name: str*) → int

libtcodpy.**parser_get_list_property**(*parser: Any*, *name: str*, *type: Any*) → Any

libtcodpy.**parser_get_string_property**(*parser: Any*, *name: str*) → str

libtcodpy.**parser_new**() → Any

libtcodpy.**parser_new_struct**(*parser: Any*, *name: str*) → Any

libtcodpy.**parser_run**(*parser: Any*, *filename: str | PathLike[str]*, *listener: Any = None*) → None

20.15 random

libtcodpy.**random_delete**(*rnd:* Random) → None

Does nothing. libtcod objects are managed by Python's garbage collector.

This function exists for backwards compatibility with libtcodpy.

libtcodpy.**random_get_double**(*rnd:* Random | *None*, *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 functions return a double precision float.

libtcodpy.**random_get_double_mean**(*rnd:* Random | *None*, *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 functions return a double precision float.

libtcodpy.**random_get_float**(*rnd:* Random | *None*, *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.
- **mi** (*float*) – The lower bound of the random range, inclusive.
- **ma** (*float*) – The upper bound of the random range, inclusive.

Returns

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

Return type
float

libtcodpy.**random_get_float_mean**(*rnd:* Random | *None*, *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.
- **mi** (*float*) – The lower bound of the random range, inclusive.
- **ma** (*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

libtcodpy.**random_get_instance**() → *Random*

Return the default Random instance.

Returns
A Random instance using the default random number generator.

Return type
Random

libtcodpy.**random_get_int**(*rnd:* Random | *None*, *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.
- **mi** (*int*) – The lower bound of the random range, inclusive.
- **ma** (*int*) – The upper bound of the random range, inclusive.

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

Return type
int

libtcodpy.**random_get_int_mean**(*rnd:* Random | *None*, *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.
- **mi** (*int*) – The lower bound of the random range, inclusive.
- **ma** (*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

libtcodpy.**random_new**(*algo: int = 1*) → *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

libtcodpy.**random_new_from_seed**(*seed: Hashable*, *algo: int = 1*) → *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

libtcodpy.**random_restore**(*rnd: Random | None*, *backup: 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.

libtcodpy.**random_save**(*rnd: Random | None*) → *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.

libtcodpy.**random_set_distribution**(*rnd: Random | None*, *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_*.

20.16 struct

libtcodpy.**struct_add_flag**(*struct: Any*, *name: str*) → None

libtcodpy.**struct_add_list_property**(*struct: Any*, *name: str*, *typ: int*, *mandatory: bool*) → None

libtcodpy.**struct_add_property**(*struct: Any*, *name: str*, *typ: int*, *mandatory: bool*) → None

libtcodpy.**struct_add_structure**(*struct: Any*, *sub_struct: Any*) → None

libtcodpy.**struct_add_value_list**(*struct: Any*, *name: str*, *value_list: Iterable[str]*, *mandatory: bool*) → None

libtcodpy.**struct_get_name**(*struct: Any*) → str

libtcodpy.**struct_get_type**(*struct: Any*, *name: str*) → int

libtcodpy.**struct_is_mandatory**(*struct: Any*, *name: str*) → bool

20.17 other

class libtcodpy.**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: 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

Clear 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() → *ConsoleBuffer*

Return 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 libtcodpy.**Dice**(*nb_dices=0*, *nb_faces=0*, *multiplier=0*, *addsub=0*)

A libtcod dice object.

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

TWENTYONE

SDL AUDIO TCOD.SDL.AUDIO

SDL2 audio playback and recording tools.

This module includes SDL's low-level audio API and a naive implementation of an SDL mixer. If you have experience with audio mixing then you might be better off writing your own mixer or modifying the existing one which was written using Python/Numpy.

This module is designed to integrate with the wider Python ecosystem. It leaves the loading to sound samples to other libraries like SoundFile.

Example:

```
# Synchronous audio example using SDL's low-level API.
import time

import soundfile  # pip install soundfile
import tcod.sdl.audio

device = tcod.sdl.audio.open()  # Open the default output device.
sound, sample_rate = soundfile.read("example_sound.wav", dtype="float32")  # Load an␣
↪audio sample using SoundFile.
converted = device.convert(sound, sample_rate)  # Convert this sample to the format␣
↪expected by the device.
device.queue_audio(converted)  # Play audio synchronously by appending it to the device␣
↪buffer.

while device.queued_samples:  # Wait until device is done playing.
    time.sleep(0.001)
```

Example:

```
# Asynchronous audio example using BasicMixer.
import time

import soundfile  # pip install soundfile
import tcod.sdl.audio

mixer = tcod.sdl.audio.BasicMixer(tcod.sdl.audio.open())  # Setup BasicMixer with the␣
↪default audio output.
sound, sample_rate = soundfile.read("example_sound.wav")  # Load an audio sample using␣
↪SoundFile.
sound = mixer.device.convert(sound, sample_rate)  # Convert this sample to the format␣
↪expected by the device.
```

(continues on next page)

(continued from previous page)

```
channel = mixer.play(sound)  # Start asynchronous playback, audio is mixed on a separate␣
↪Python thread.
while channel.busy:  # Wait until the sample is done playing.
    time.sleep(0.001)
```

New in version 13.5.

class tcod.sdl.audio.**AllowedChanges**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Which parameters are allowed to be changed when the values given are not supported.

ANY = 15

CHANNELS = 4

FORMAT = 2

FREQUENCY = 1

NONE = 0

SAMPLES = 8

class tcod.sdl.audio.**AudioDevice**(*device_id: int*, *capture: bool*, *spec: Any*)

An SDL audio device.

Open new audio devices using *tcod.sdl.audio.open*.

When you use this object directly the audio passed to *queue_audio* is always played synchronously. For more typical asynchronous audio you should pass an AudioDevice to *BasicMixer*.

Changed in version 16.0: Can now be used as a context which will close the device on exit.

__enter__() → Self

Return self and enter a managed context.

__exit__(*type: type[BaseException] | None*, *value: BaseException | None*, *traceback: TracebackType | None*) → None

Close the device when exiting the context.

__repr__() → str

Return a representation of this device.

close() → None

Close this audio device. Using this object after it has been closed is invalid.

convert(*sound: ArrayLike*, *rate: int | None = None*) → NDArray[Any]

Convert an audio sample into a format supported by this device.

Returns the converted array. This might be a reference to the input array if no conversion was needed.

Parameters

- **sound** – An ArrayLike sound sample.
- **rate** – The sample-rate of the input array. If None is given then it's assumed to be the same as the device.

New in version 13.6.

See also:

convert_audio

dequeue_audio() → NDArray[Any]

Return the audio buffer from a capture stream.

queue_audio(*samples: numpy.typing.ArrayLike*) → None

Append audio samples to the audio data queue.

buffer_bytes: Final[int]

The size of the audio buffer in bytes.

buffer_samples: Final[int]

The size of the audio buffer in samples.

property callback: Callable[[*AudioDevice*, NDArray[Any]], None]

If the device was opened with a callback enabled, then you may get or set the callback with this attribute.

channels: Final[int]

The number of audio channels for this device.

device_id: Final[int]

The SDL device identifier used for SDL C functions.

format: Final[np.dtype[Any]]

The format used for audio samples with this device.

frequency: Final[int]

The audio device sound frequency.

is_capture: Final[bool]

True if this is a recording device instead of an output device.

property paused: bool

Get or set the device paused state.

property queued_samples: int

The current amount of samples remaining in the audio queue.

silence: float

The value of silence, according to SDL.

spec: Final[Any]

The SDL_AudioSpec as a CFFI object.

property stopped: bool

Is True if the device has failed or was closed.

class tcod.sdl.audio.**BasicMixer**(*device:* AudioDevice)

An SDL sound mixer implemented in Python and Numpy.

New in version 13.6.

close() → None

Shutdown this mixer, all playing audio will be abruptly stopped.

get_channel(*key: Hashable*) → *Channel*

Return a channel tied to with the given key.

Channels are initialized as you access them with this function. `int` channels starting from zero are used internally.

This can be used to generate a `"music"` channel for example.

play(*sound: numpy.typing.ArrayLike*, *, *volume: float | tuple[float, ...] = 1.0*, *loops: int = 0*, *on_end: Callable[[Channel], None] | None = None*) → *Channel*

Play a sound, return the channel the sound is playing on.

Parameters

- **sound** – The sound to play. This a Numpy array matching the format of the loaded audio device.
- **volume** – The volume to play the sound at. You can also pass a tuple of floats to set the volume for each channel/speaker.
- **loops** – How many times to play the sound, *-1* can be used to loop the sound forever.
- **on_end** – A function to call when this sound has ended. This is called with the *Channel* which was playing the sound.

run() → None

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop() → None

Stop playback on all channels from this mixer.

device

The *AudioDevice*

class tcod.sdl.audio.**Channel**

An audio channel for *BasicMixer*. Use *BasicMixer.get_channel* to initialize this object.

New in version 13.6.

fadeout(*time: float*) → None

Fadeout this channel then stop playing.

play(*sound: numpy.typing.ArrayLike*, *, *volume: float | tuple[float, ...] = 1.0*, *loops: int = 0*, *on_end: Callable[[Channel], None] | None = None*) → None

Play an audio sample, stopping any audio currently playing on this channel.

Parameters are the same as *BasicMixer.play*.

stop() → None

Stop audio on this channel.

property busy: bool

Is True when this channel is playing audio.

mixer: *BasicMixer*

The *BasicMixer* is channel belongs to.

tcod.sdl.audio.**convert_audio**(*in_sound: ArrayLike*, *in_rate: int*, *, *out_rate: int*, *out_format: DTypeLike*, *out_channels: int*) → NDArray[Any]

Convert an audio sample into a format supported by this device.

Returns the converted array. This might be a reference to the input array if no conversion was needed.

Parameters

- **in_sound** – The input ArrayLike sound sample. Input format and channels are derived from the array.
- **in_rate** – The sample-rate of the input array.
- **out_rate** – The sample-rate of the output array.
- **out_format** – The output format of the converted array.
- **out_channels** – The number of audio channels of the output array.

New in version 13.6.

Changed in version 16.0: Now converts floating types to *np.float32* when SDL doesn't support the specific format.

See also:

AudioDevice.convert

tcod.sdl.audio.**get_capture_devices**() → Iterator[str]

Iterate over the available audio capture devices.

tcod.sdl.audio.**get_devices**() → Iterator[str]

Iterate over the available audio output devices.

tcod.sdl.audio.**open**(*name: str | None = None*, *capture: bool = False*, *, *frequency: int = 44100*, *format: DTypeLike = <class 'numpy.float32'>*, *channels: int = 2*, *samples: int = 0*, *allowed_changes: AllowedChanges = AllowedChanges.NONE*, *paused: bool = False*, *callback: None | Literal[True] | Callable[[AudioDevice, NDArray[Any]], None] = None*) → *AudioDevice*

Open an audio device for playback or capture and return it.

Parameters

- **name** – The name of the device to open, or None for the most reasonable default.
- **capture** – True if this is a recording device, or False if this is an output device.
- **frequency** – The desired sample rate to open the device with.
- **format** – The data format to use for samples as a NumPy dtype.
- **channels** – The number of speakers for the device. 1, 2, 4, or 6 are typical options.
- **samples** – The desired size of the audio buffer, must be a power of two.
- **allowed_changes** – By default if the hardware does not support the desired format than SDL will transparently convert between formats for you. Otherwise you can specify which parameters are allowed to be changed to fit the hardware better.
- **paused** – If True then the device will begin in a paused state. It can then be unpaused by assigning False to *AudioDevice.paused*.
- **callback** – If None then this device will be opened in push mode and you'll have to use *AudioDevice.queue_audio* to send audio data or *AudioDevice.dequeue_audio* to receive it. If a callback is given then you can change it later, but you can not enable or disable

the callback on an opened device. If True then a default callback which plays silence will be used, this is useful if you need the audio device before your callback is ready.

If a callback is given then it will be called with the *AudioDevice* and a Numpy buffer of the data stream. This callback will be run on a separate thread. Exceptions not handled by the callback become unraiseable and will be handled by `sys.unraisablehook`.

See also:

https://wiki.libsdl.org/SDL_AudioSpec https://wiki.libsdl.org/SDL_OpenAudioDevice

CHAPTER

TWENTYTWO

SDL JOYSTICK SUPPORT tcod.sdl.joystick

SDL Joystick Support.

New in version 13.8.

class tcod.sdl.joystick.**ControllerAxis**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The standard axes for a game controller.

LEFTX = 0

LEFTY = 1

RIGHTX = 2

RIGHTY = 3

TRIGGERLEFT = 4

TRIGGERRIGHT = 5

class tcod.sdl.joystick.**ControllerButton**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The standard buttons for a game controller.

A = 0

B = 1

BACK = 4

DPAD_DOWN = 12

DPAD_LEFT = 13

DPAD_RIGHT = 14

DPAD_UP = 11

GUIDE = 5

LEFTSHOULDER = 9

LEFTSTICK = 7

MISC1 = 15

PADDLE1 = 16

PADDLE2 = 17

PADDLE3 = 18

PADDLE4 = 19

RIGHTSHOULDER = 10

RIGHTSTICK = 8

START = 6

TOUCHPAD = 20

X = 2

Y = 3

class tcod.sdl.joystick.**GameController**(*sdl_controller_p: Any*)

A standard interface for an Xbox 360 style game controller.

get_axis(*axis:* ControllerAxis) → int

Return the state of the given *axis*.

The state is usually a value from -32768 to 32767, with positive values towards the lower-right direction. Triggers have the range of 0 to 32767 instead.

get_button(*button:* ControllerButton) → bool

Return True if *button* is currently held.

joystick: **Final**

The *Joystick* associated with this controller.

class tcod.sdl.joystick.**Joystick**(*sdl_joystick_p: Any*)

A low-level SDL joystick.

See also:

https://wiki.libsdl.org/CategoryJoystick

get_axis(*axis: int*) → int

Return the raw value of *axis* in the range -32768 to 32767.

get_ball(*ball: int*) → tuple[int, int]

Return the values (delta_x, delta_y) of *ball* since the last poll.

get_button(*button: int*) → bool

Return True if *button* is currently held.

get_current_power() → *Power*

Return the power level/state of this joystick. See *Power*.

get_hat(*hat: int*) → tuple[~typing.Literal[-1, 0, 1], ~typing.Literal[-1, 0, 1]]

Return the direction of *hat* as (x, y). With (-1, -1) being in the upper-left.

axes: **Final[int]**

The total number of axes.

balls: Final[int]

The total number of trackballs.

buttons: Final[int]

The total number of buttons.

guid: Final[str]

The GUID of this joystick.

hats: Final[int]

The total number of hats.

id: Final[int]

The instance ID of this joystick. This is not the same as the device ID.

name: Final[str]

The name of this joystick.

sdl_joystick_p: Final

The CFFI pointer to an SDL_Joystick struct.

tcod.sdl.joystick.**controller_event_state**(*new_state: bool | None = None*) → bool

Check or set game controller event polling.

See also:

https://wiki.libsdl.org/SDL_GameControllerEventState

tcod.sdl.joystick.**get_controllers**() → list[*GameController*]

Return a list of all connected game controllers.

This ignores joysticks without a game controller mapping.

tcod.sdl.joystick.**get_joysticks**() → list[*Joystick*]

Return a list of all connected joystick devices.

tcod.sdl.joystick.**init**() → None

Initialize SDL's joystick and game controller subsystems.

tcod.sdl.joystick.**joystick_event_state**(*new_state: bool | None = None*) → bool

Check or set joystick event polling.

See also:

https://wiki.libsdl.org/SDL_JoystickEventState

class tcod.sdl.joystick.**Power**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

The possible power states of a controller.

See also:

Joystick.get_current_power

UNKNOWN = -1

Power state is unknown.

EMPTY = 0

<= 5% power.

LOW = 1

<= 20% power.

MEDIUM = 2

<= 70% power.

FULL = 3

<= 100% power.

WIRED = 4

MAX = 5

CHAPTER

TWENTYTHREE

SDL RENDERING TCOD.SDL.RENDER

SDL2 Rendering functionality.

New in version 13.4.

class tcod.sdl.render.**BlendFactor**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

SDL blend factors.

See also:

compose_blend_mode https://wiki.libsdl.org/SDL_BlendFactor

New in version 13.5.

DST_ALPHA = 9

DST_COLOR = 7

ONE = 2

ONE_MINUS_DST_ALPHA = 10

ONE_MINUS_DST_COLOR = 8

ONE_MINUS_SRC_ALPHA = 6

ONE_MINUS_SRC_COLOR = 4

SRC_ALPHA = 5

SRC_COLOR = 3

ZERO = 1

class tcod.sdl.render.**BlendMode**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

SDL blend modes.

See also:

Texture.blend_mode *Renderer.draw_blend_mode* *compose_blend_mode*

New in version 13.5.

ADD = 2

BLEND = 1

INVALID = 2147483647

MOD = 4

NONE = 0

class tcod.sdl.render.**BlendOperation**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

SDL blend operations.

See also:

compose_blend_mode https://wiki.libsdl.org/SDL_BlendOperation

New in version 13.5.

ADD = 1

dest + source

MAXIMUM = 5

max(dest, source)

MINIMUM = 4

min(dest, source)

REV_SUBTRACT = 3

source - dest

SUBTRACT = 2

dest - source

class tcod.sdl.render.**Renderer**(*sdl_renderer_p: Any*)

SDL Renderer.

__eq__(*other: object*) → bool

Return True if compared to the same renderer.

clear() → None

Clear the current render target with *draw_color*.

New in version 13.5.

copy(*texture:* Texture, *source: tuple[float, float, float, float] | None = None*, *dest: tuple[float, float, float, float] | None = None*, *angle: float = 0*, *center: tuple[float, float] | None = None*, *flip:* RendererFlip *= RendererFlip.NONE*) → None

Copy a texture to the rendering target.

Parameters

- **texture** – The texture to copy onto the current texture target.
- **source** – The (x, y, width, height) region of *texture* to copy. If None then the entire texture is copied.
- **dest** – The (x, y, width, height) region of the target. If None then the entire target is drawn over.
- **angle** – The angle in degrees to rotate the image clockwise.
- **center** – The (x, y) point where rotation is applied. If None then the center of *dest* is used.
- **flip** – Flips the *texture* when drawing it.

Changed in version 13.5: *source* and *dest* can now be float tuples. Added the *angle*, *center*, and *flip* parameters.

draw_line(*start: tuple[float, float]*, *end: tuple[float, float]*) → None

Draw a single line.

New in version 13.5.

draw_lines(*points: NDArray[np.intc | np.float32]*) → None

Draw a connected series of lines from an array.

New in version 13.5.

draw_point(*xy: tuple[float, float]*) → None

Draw a point.

New in version 13.5.

draw_points(*points: NDArray[np.intc | np.float32]*) → None

Draw an array of points.

New in version 13.5.

draw_rect(*rect: tuple[float, float, float, float]*) → None

Draw a rectangle outline.

New in version 13.5.

draw_rects(*rects: NDArray[np.intc | np.float32]*) → None

Draw multiple outlined rectangles from an array.

New in version 13.5.

fill_rect(*rect: tuple[float, float, float, float]*) → None

Fill a rectangle with `draw_color`.

New in version 13.5.

fill_rects(*rects: NDArray[np.intc | np.float32]*) → None

Fill multiple rectangles from an array.

New in version 13.5.

geometry(*texture: Texture | None*, *xy: NDArray[np.float32]*, *color: NDArray[np.uint8]*, *uv: NDArray[np.float32]*, *indices: NDArray[np.uint8 | np.uint16 | np.uint32] | None = None*) → None

Render triangles from texture and vertex data.

New in version 13.5.

new_texture(*width: int*, *height: int*, *, *format: int | None = None*, *access: int | None = None*) → *Texture*

Allocate and return a new Texture for this renderer.

Parameters

- **width** – The pixel width of the new texture.
- **height** – The pixel height of the new texture.
- **format** – The format the new texture.
- **access** – The access mode of the texture. Defaults to `TextureAccess.STATIC`. See `TextureAccess` for more options.

present() → None

Present the currently rendered image to the screen.

read_pixels(*, *rect: tuple[int, int, int, int] | None = None*, *format: int | Literal['RGB', 'RGBA'] = 'RGBA'*, *out: NDArray[np.uint8] | None = None*) → NDArray[np.uint8]

Fetch the pixel contents of the current rendering target to an array.

By default returns an RGBA pixel array of the full target in the shape: `(height, width, rgba)`. The target can be changed with *set_render_target*

Parameters

- **rect** – The `(left, top, width, height)` region of the target to fetch, or None for the entire target.
- **format** – The pixel format. Defaults to `"RGBA"`.
- **out** – The output array. Can be None or must be an `np.uint8` array of shape: `(height, width, channels)`. Must be C contiguous along the `(width, channels)` axes.

This operation is slow due to coping from VRAM to RAM. When reading the main rendering target this should be called after rendering and before *present*. See https://wiki.libsdl.org/SDL2/SDL_RenderReadPixels

Returns
`(height, width, channels)` with the fetched pixels.

Return type
The output uint8 array of shape

New in version 15.0.

set_render_target(*texture: Texture*) → _RestoreTargetContext

Change the render target to *texture*, returns a context that will restore the original target when exited.

set_vsync(*enable: bool*) → None

Enable or disable VSync for this renderer.

New in version 13.5.

upload_texture(*pixels: NDArray[Any]*, *, *format: int | None = None*, *access: int | None = None*) → *Texture*

Return a new Texture from an array of pixels.

Parameters

- **pixels** – An RGB or RGBA array of pixels in row-major order.
- **format** – The format of *pixels* when it isn't a simple RGB or RGBA array.
- **access** – The access mode of the texture. Defaults to *TextureAccess.STATIC*. See *TextureAccess* for more options.

property clip_rect: tuple[int, int, int, int] | None

Get or set the clipping rectangle of this renderer.

Set to None to disable clipping.

New in version 13.5.

property draw_blend_mode: *BlendMode*

Get or set the active blend mode of this renderer.

New in version 13.5.

property draw_color: `tuple[int, int, int, int]`

Get or set the active RGBA draw color for this renderer.

New in version 13.5.

property integer_scaling: `bool`

Get or set if this renderer enforces integer scaling.

See also:

https://wiki.libsdl.org/SDL_RenderSetIntegerScale

New in version 13.5.

property logical_size: `tuple[int, int]`

Get or set a device independent (width, height) resolution.

Might be (0, 0) if a resolution was never assigned.

See also:

https://wiki.libsdl.org/SDL_RenderSetLogicalSize

New in version 13.5.

property output_size: `tuple[int, int]`

Get the (width, height) pixel resolution of the rendering context.

See also:

https://wiki.libsdl.org/SDL_GetRendererOutputSize

New in version 13.5.

property scale: `tuple[float, float]`

Get or set an (x_scale, y_scale) multiplier for drawing.

See also:

https://wiki.libsdl.org/SDL_RenderSetScale

New in version 13.5.

property viewport: `tuple[int, int, int, int] | None`

Get or set the drawing area for the current rendering target.

See also:

https://wiki.libsdl.org/SDL_RenderSetViewport

New in version 13.5.

class tcod.sdl.render.**RendererFlip**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Flip parameter for *Renderer.copy*.

HORIZONTAL = 1

Flip the image horizontally.

NONE = 0

Default value, no flip.

VERTICAL = 2

Flip the image vertically.

class tcod.sdl.render.**Texture**(*sdl_texture_p: Any*, *sdl_renderer_p: Any* = *None*)

SDL hardware textures.

Create a new texture using *Renderer.new_texture* or *Renderer.upload_texture*.

__eq__(*other: object*) → bool

Return True if compared to the same texture.

update(*pixels: NDArray[Any]*, *rect: tuple[int, int, int, int] | None* = *None*) → None

Update the pixel data of this texture.

New in version 13.5.

access: Final[*TextureAccess*]

Texture access mode, read only.

Changed in version 13.5: Attribute is now a *TextureAccess* value.

property alpha_mod: int

Texture alpha modulate value, can be set to 0 - 255.

property blend_mode: *BlendMode*

Texture blend mode, can be set.

Changed in version 13.5: Property now returns a BlendMode instance.

property color_mod: tuple[int, int, int]

Texture RGB color modulate values, can be set.

format: Final[int]

Texture format, read only.

height: Final[int]

Texture pixel height, read only.

width: Final[int]

Texture pixel width, read only.

class tcod.sdl.render.**TextureAccess**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Determines how a texture is expected to be used.

STATIC = 0

Texture rarely changes.

STREAMING = 1

Texture frequently changes.

TARGET = 2

Texture will be used as a render target.

tcod.sdl.render.**compose_blend_mode**(*source_color_factor:* BlendFactor, *dest_color_factor:* BlendFactor, *color_operation:* BlendOperation, *source_alpha_factor:* BlendFactor, *dest_alpha_factor:* BlendFactor, *alpha_operation:* BlendOperation) → *BlendMode*

Return a custom blend mode composed of the given factors and operations.

See also:

https://wiki.libsdl.org/SDL_ComposeCustomBlendMode

New in version 13.5.

tcod.sdl.render.**new_renderer**(*window: Window*, *, *driver: int | None = None*, *software: bool = False*, *vsync: bool = True*, *target_textures: bool = False*) → *Renderer*

Initialize and return a new SDL Renderer.

Parameters

- **window** – The window that this renderer will be attached to.
- **driver** – Force SDL to use a specific video driver.
- **software** – If True then a software renderer will be forced. By default a hardware renderer is used.
- **vsync** – If True then Vsync will be enabled.
- **target_textures** – If True then target textures can be used by the renderer.

Example:

```
# Start by creating a window.
sdl_window = tcod.sdl.video.new_window(640, 480)
# Create a renderer with target texture support.
sdl_renderer = tcod.sdl.render.new_renderer(sdl_window, target_textures=True)
```

See also:

tcod.sdl.video.new_window()

CHAPTER

TWENTYFOUR

SDL MOUSE FUNCTIONS `TCOD.SDL.MOUSE`

SDL mouse and cursor functions.

You can use this module to move or capture the cursor.

You can also set the cursor icon to an OS-defined or custom icon.

New in version 13.5.

class tcod.sdl.mouse.**Cursor**(*sdl_cursor_p: Any*)

A cursor icon for use with `set_cursor`.

class tcod.sdl.mouse.**SystemCursor**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

An enumerator of system cursor icons.

ARROW = 0

CROSSHAIR = 3

HAND = 11

IBEAM = 1

NO = 10

SIZEALL = 9

SIZENESW = 6

SIZENS = 8

SIZENWSE = 5

SIZEWE = 7

WAIT = 2

WAITARROW = 4

tcod.sdl.mouse.**capture**(*enable: bool*) → None

Enable or disable mouse capture to track the mouse outside of a window.

It is highly recommended to read the related remarks section in the SDL docs before using this.

Example:

```
# Make mouse button presses capture the mouse until all buttons are released.
# This means that dragging the mouse outside of the window will not cause an␣
↪interruption in motion events.
for event in tcod.event.get():
    match event:
        case tcod.event.MouseButtonDown(button=button, pixel=pixel):  # Clicking␣
↪the window captures the mouse.
            tcod.sdl.mouse.capture(True)
        case tcod.event.MouseButtonUp():  # When all buttons are released then the␣
↪mouse is released.
            if tcod.event.mouse.get_global_state().state == 0:
                tcod.sdl.mouse.capture(False)
        case tcod.event.MouseMotion(pixel=pixel, pixel_motion=pixel_motion,␣
↪state=state):
            pass  # While a button is held this event is still captured outside of␣
↪the window.
```

See also:

tcod.sdl.mouse.set_relative_mode https://wiki.libsdl.org/SDL_CaptureMouse

tcod.sdl.mouse.**get_cursor**() → *Cursor* | None

Return the active cursor, or None if these is no mouse.

tcod.sdl.mouse.**get_default_cursor**() → *Cursor*

Return the default cursor.

tcod.sdl.mouse.**get_focus**() → *Window* | None

Return the window which currently has mouse focus.

tcod.sdl.mouse.**get_global_state**() → MouseState

Return the mouse state relative to the desktop.

See also:

https://wiki.libsdl.org/SDL_GetGlobalMouseState

tcod.sdl.mouse.**get_relative_mode**() → bool

Return True if relative mouse mode is enabled.

tcod.sdl.mouse.**get_relative_state**() → MouseState

Return the mouse state, the coordinates are relative to the last time this function was called.

See also:

https://wiki.libsdl.org/SDL_GetRelativeMouseState

tcod.sdl.mouse.**get_state**() → MouseState

Return the mouse state relative to the window with mouse focus.

See also:

https://wiki.libsdl.org/SDL_GetMouseState

tcod.sdl.mouse.**new_color_cursor**(*pixels: numpy.typing.ArrayLike*, *hot_xy: tuple[int, int]*) → *Cursor*

Create a new color cursor.

Parameters

- **pixels** – A row-major array of RGB or RGBA pixels.

- **hot_xy** – The position of the pointer relative to the mouse sprite, starting from the upper-left at (0, 0).

See also:

set_cursor

tcod.sdl.mouse.**new_cursor**(*data: NDArray[np.bool_]*, *mask: NDArray[np.bool_]*, *hot_xy: tuple[int, int] = (0, 0)*) → *Cursor*

Return a new non-color Cursor from the provided parameters.

Parameters

- **data** – A row-major boolean array for the data parameters. See the SDL docs for more info.
- **mask** – A row-major boolean array for the mask parameters. See the SDL docs for more info.
- **hot_xy** – The position of the pointer relative to the mouse sprite, starting from the upper-left at (0, 0).

See also:

set_cursor https://wiki.libsdl.org/SDL_CreateCursor

tcod.sdl.mouse.**new_system_cursor**(*cursor:* SystemCursor) → *Cursor*

Return a new Cursor from one of the system cursors labeled by SystemCursor.

See also:

set_cursor

tcod.sdl.mouse.**set_cursor**(*cursor:* Cursor | SystemCursor | *None*) → None

Change the active cursor to the one provided.

Parameters

cursor – A cursor created from *new_cursor*, *new_color_cursor*, or *new_system_cursor*. Can also take values of *SystemCursor* directly. None will force the current cursor to be redrawn.

tcod.sdl.mouse.**set_relative_mode**(*enable: bool*) → None

Enable or disable relative mouse mode which will lock and hide the mouse and only report mouse motion.

See also:

tcod.sdl.mouse.capture https://wiki.libsdl.org/SDL_SetRelativeMouseMode

tcod.sdl.mouse.**show**(*visible: bool | None = None*) → bool

Optionally show or hide the mouse cursor then return the state of the cursor.

Parameters

visible – If None then only return the current state. Otherwise set the mouse visibility.

Returns

True if the cursor is visible.

New in version 16.0.

tcod.sdl.mouse.**warp_global**(*x: int*, *y: int*) → None

Move the mouse cursor to a position on the desktop.

tcod.sdl.mouse.**warp_in_window**(*window:* Window, *x: int*, *y: int*) → None

Move the mouse cursor to a position within a window.

CHAPTER

TWENTYFIVE

SDL WINDOW AND DISPLAY API `TCOD.SDL.VIDEO`

SDL2 Window and Display handling.

There are two main ways to access the SDL window. Either you can use this module to open a window yourself bypassing libtcod's context, or you can use `Context.sdl_window` to get the window being controlled by that context (if the context has one.)

New in version 13.4.

class tcod.sdl.video.**FlashOperation**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Values for `Window.flash`.

BRIEFLY = 1

Flash briefly.

CANCEL = 0

Stop flashing.

UNTIL_FOCUSED = 2

Flash until focus is gained.

class tcod.sdl.video.**Window**(*sdl_window_p: Any*)

An SDL2 Window object.

flash(*operation:* FlashOperation = *FlashOperation.UNTIL_FOCUSED*) → None

Get the users attention.

hide() → None

Hide this window.

maximize() → None

Make the window as big as possible.

minimize() → None

Minimize the window to an iconic state.

raise_window() → None

Raise the window and set input focus.

restore() → None

Restore a minimized or maximized window to its original size and position.

set_icon(*pixels: numpy.typing.ArrayLike*) → None

Set the window icon from an image.

Parameters
pixels – A row-major array of RGB or RGBA pixel values.

show() → None
Show this window.

property border_size: tuple[int, int, int, int]
Get the (top, left, bottom, right) size of the window decorations around the client area.

If this fails or the window doesn't have decorations yet then the value will be (0, 0, 0, 0).

See also:

https://wiki.libsdl.org/SDL_GetWindowBordersSize

property flags: *WindowFlags*
The current flags of this window, read-only.

property fullscreen: int
Get or set the fullscreen status of this window.

Can be set to the *WindowFlags.FULLSCREEN* or *WindowFlags.FULLSCREEN_DESKTOP* flags.

Example:

```
# Toggle fullscreen.
window: tcod.sdl.video.Window
if window.fullscreen:
    window.fullscreen = False  # Set windowed mode.
else:
    window.fullscreen = tcod.sdl.video.WindowFlags.FULLSCREEN_DESKTOP
```

property grab: bool
Get or set this windows input grab mode.

See also:

https://wiki.libsdl.org/SDL_SetWindowGrab

property max_size: tuple[int, int]
Get or set this windows maximum client area.

property min_size: tuple[int, int]
Get or set this windows minimum client area.

property mouse_rect: tuple[int, int, int, int] | None
Get or set the mouse confinement area when the window has mouse focus.

Setting this will not automatically grab the cursor.

New in version 13.5.

property opacity: float
Get or set this windows opacity. 0.0 is fully transparent and 1.0 is fully opaque.

Will error if you try to set this and opacity isn't supported.

property position: tuple[int, int]
Get or set the (x, y) position of the window.

This attribute can be set the move the window. The constants tcod.lib.SDL_WINDOWPOS_CENTERED or tcod.lib.SDL_WINDOWPOS_UNDEFINED may be used.

property resizable: `bool`

Get or set if this window can be resized.

property size: `tuple[int, int]`

Get or set the pixel (width, height) of the window client area.

This attribute can be set to change the size of the window but the given size must be greater than (1, 1) or else ValueError will be raised.

property title: `str`

Get or set the title of the window.

class tcod.sdl.video.**WindowFlags**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bit flags which make up a windows state.

See also:

https://wiki.libsdl.org/SDL_WindowFlags

ALLOW_HIGHDPI = 8192

ALWAYS_ON_TOP = 32768

BORDERLESS = 16

FOREIGN = 2048

FULLSCREEN = 1

FULLSCREEN_DESKTOP = 4097

HIDDEN = 8

INPUT_FOCUS = 512

MAXIMIZED = 128

METAL = 536870912

MINIMIZED = 64

MOUSE_CAPTURE = 16384

MOUSE_FOCUS = 1024

MOUSE_GRABBED = 256

OPENGL = 2

POPUP_MENU = 524288

RESIZABLE = 32

SHOWN = 4

SKIP_TASKBAR = 65536

TOOLTIP = 262144

UTILITY = 131072

VULKAN = 268435456

tcod.sdl.video.**get_grabbed_window**() → *Window* | None

Return the window which has input grab enabled, if any.

tcod.sdl.video.**new_window**(*width: int*, *height: int*, ***, *x: int* | *None* = *None*, *y: int* | *None* = *None*, *title: str* | *None* = *None*, *flags: int* = *0*) → *Window*

Initialize and return a new SDL Window.

Parameters

- **width** – The requested pixel width of the window.
- **height** – The requested pixel height of the window.
- **x** – The left-most position of the window.
- **y** – The top-most position of the window.
- **title** – The title text of the new window. If no option is given then *sys.arg[0]* will be used as the title.
- **flags** – The SDL flags to use for this window, such as *tcod.sdl.video.WindowFlags.RESIZABLE*. See *WindowFlags* for more options.

Example:

```
import tcod.sdl.video
# Create a new resizable window with a custom title.
window = tcod.sdl.video.new_window(640, 480, title="Title bar text", flags=tcod.sdl.
↪video.WindowFlags.RESIZABLE)
```

See also:

tcod.sdl.render.new_renderer()

tcod.sdl.video.**screen_saver_allowed**(*allow: bool* | *None* = *None*) → bool

Allow or prevent a screen saver from being displayed and return the current allowed status.

If *allow* is *None* then only the current state is returned. Otherwise it will change the state before checking it.

SDL typically disables the screensaver by default. If you're unsure, then don't touch this.

Example:

```
import tcod.sdl.video

print(f"Screen saver was allowed: {tcod.sdl.video.screen_saver_allowed()}")
# Allow the screen saver.
# Might be okay for some turn-based games which don't use a gamepad.
tcod.sdl.video.screen_saver_allowed(True)
```

CHAPTER

TWENTYSIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z