Earwax

Feb 16, 2021

Contents:

CHAPTER 1

Introduction

1.1 Project Goals

Earwax is an audio game library with a focus on readable code, minimal boilerplate, and rapid prototyping.

It should be possible to create a basic game with basic code. It should also be possible to add layers of complexity without the game library holding you back.

1.2 Workflow

The basic flow of an Earwax program is:

- Create a *Game instance*.
- Create 1 or more *Level* instances.
- Add actions to the level instance(s) you created in the previous step.
- Create a pyglet *Window* instance.
- Run the game object you created in step ‘ with the window object you created in the previous step.

1.3 Full Example

The below code is a full -albeit minimal - code example:

```
from earwax import Game, Level
from pyglet.window import key, mouse, Window
w = Window(caption='Test Game')
g = Game()
l = Level(g)

```

(continues on next page)

(continued from previous page)

```python

@l.action('Key speak', symbol=key.S)
def key_speak():
    """Say something when the s key is pressed."""
    g.output('You pressed the s key.')


@l.action('Mouse speak', mouse_button=mouse.LEFT)
def mouse_speak():
    """Speak when the left mouse button is pressed."""
    g.output('You pressed the left mouse button.')


@l.action('Quit', symbol=key.ESCAPE, mouse_button=mouse.RIGHT)
def do_quit():
    """Quit the game."""
    g.stop()


g.run(w, initial_level=l)
```

CHAPTER 2

Installation

CHAPTER 3

Installing Using pip

It is recommended that you install Earwax using pip:

```
pip install Earwax
```

CHAPTER 4

Install Using Git

Alternatively, you could install using git:

```
git clone https://github.com/chrisnorman7/earwax.git
cd earwax
python setup.py
```

CHAPTER 5

Running Tests

To run the tests, you will need to install pytest:

```
pip install pytest
```

Then to run the tests:

```
py.test
```

While the tests run, many windows will appear and disappear. That is completely normal, I just use lots of Pyglet for testing.

CHAPTER 6

Building Documentation

You can always find the most up to date version of the docs on Read the Docs, but you can also build them yourself:

```
pip install -Ur docs/requirements.txt
python setup.py build_sphinx
```

CHAPTER 7

Features

7.1 Implemented Features

- Ability to separate disparate parts of a game into `Level` constructs.
- Ability to push, pop, and replace Level instances on the central `Game` object.
- Uses Pyglet's event system, mostly eliminating the need to subclass.
- Uses Synthizer as its sound backend.
- Both `simple` and `advanced` sound players, designed for playing interface sounds.
- A flexible and unobtrusive configuration framework that uses yaml.
- The ability to configure various aspects of the framework (including generic sound icons in menus), simply by setting configuration values on a `configuration object` which resides on your `game object`.
- Various functions for playing sounds and cleaning them up when they're finished.
- Different types of levels already implemented:
  - Game board levels, so you can create board games with minimal boilerplate.
  - Box levels, which contain boxes, which can be connected together to make maps. Both free and restricted movement commands are already implemented.
- The ability to add actions to `earwax.Level` instances with keyboard keys, mouse buttons, joystick buttons, and joystick hat positions.
- A text-to-speech system which uses cytolk.
- An `earwax` command which can currently create default games.
- Various Promise-style classes for long-running tasks.

7.2 Feature Requests

If you need a feature that is not already on this list, please submit a feature request.

CHAPTER 8

Tutorials

This section contains various tutorials that will show you how to use the different parts of earwax.

8.1 Getting Started

When getting started with any new library, it is often hard to know where to start. Earwax contains many tutorials, but that doesn't help you write your first line of code.

For writing your first game, there is the `game` command:

```
$ earwax game main.py
Creating a blank game at main.py.
Done.
```

This will create you a very minimal *game*, which can already be run:

```
$ python main.py
```

This should load up a game called "New Game".

This game already has a few things to get you started:

- A main *menu*, with an entry to `play the game`, *show credits*, and *exit*.
- An initial level with a *help menu*. You can press `Q` from this level to return to the main menu.
- An extremely self-aggrandising default *credit*, mentioning Earwax, and its illustrious creator.
- Commented out lines which provide main menu, and initial level *music*.

This game serves as a starting point for your own work, and should be *expanded upon*.

8.2 Editors

In earwax, an Editor represents a simple text editor.

Editors can be used for editing single lines of text. While it is entirely possible to add a line break to the text when you create an `Editor` instance, pressing the enter key while an `Editor` instance is pushed onto your game will result in the `on_submit()` event being dispatched.

8.3 Creating An Editor

Creating an editor can be done the same way you can create most `earwax.Level` instances:

```
e: Editor = Editor(game)
```

As you can see, a `earwax.Game` instance is necessary.

You can also supply a `text` argument:

```
e: editor = Editor(game, text='Hello world')
```

The cursor will be placed at the end of the text, and it can be edited with standard operating system commands, unless you alter what motions are supported of course.

8.3.1 Motions

You can easily add extra motions, or override the default ones:

```
from pyglet.window import key

@e.motion(key.MOTION_BACKSPACE)
def backspace():
    game.output('Backspace was pressed.')
```

Now, when the backspace key is pressed, your new event will fire too.

8.4 Submitting Text

When the enter key is pressed, or a game hat is used to select "submit" (more on that later), the `earwax.Editor.submit()` method is called.

You can retrieve the text that was entered with the `on_submit()` event:

```
@e.event
def on_submit(text: str) -> None:
    print('Text entered: %r.' % text)
```

8.5 Dismissing Editors

Like Earwax `menus`, editors are dismissible by default. This can of course be changed:

```
e: Editor = Editor(game, dismissible=False)
```

Now, when the escape key is pressed, nothing happens.

8.6 Editing With The Hat

You can use a game controller to edit text. Simply use the left and right directions to move through text, and the up and down directions to select letters.

If you keep pressing the up hat, you will come to a delete option. One more up performs the deletion.

If your focus is at the end of the line, the delete option will be replaced with a "Submit" option instead. This is the same as pressing the enter key.

8.7 Sounds

Being an audio game engine, sounds are a pretty important part of what Earwax can do.

As such, many useful sound functions have been added, with more to come.

This part of the tutorial will attempt to document some of these functions, more fully than the included documentation.

8.7.1 Buffer Directories

The idea behind the `earwax.BufferDirectory` class, is that quite often we need a single directory of sounds we can pick from. This usually leads to code like the following:

```
room_ambiance = Sound('sounds/ambiances/room.wav')
station_ambiance = Sound('sounds/ambiances/station.wav')
ship_ambiance = Sound('sounds/ambiances/ship.wav')
```

This is particularly error prone, although has the benefit of letting you autocomplete variable names in your editor of choice.

Inspired by a possible future feature of Synthizer, I decided to make a small utility class for the express purpose of loading a directory of sounds. Using this class, the above code can be rewritten as:

```
from pathlib import Path

from earwax import BufferDirectory

ambiances: BufferDirectory = BufferDirectory(Path('sounds/ambiances'))

room_ambiance = 'room.wav'
station_ambiance = 'station.wav'
ship_ambiance = 'ship.wav'
```

Now you can for example get the station ambiance with the below code:

```
buffer: Buffer = ambiances.buffers[station_ambiance]
```

This is useful if for example you've moved the entire directory. Instead of performing a find and replace, you can simply change the BufferDirectory instance:

```
ambiances: BufferDirectory = BufferDirectory(Path('sounds/amb'))
```

Another common idiom is to select a random sound file from a directory. Earwax has a few sound functions with this capability already. If you pass a `Path` instance which happens to be a directory to `earwax.play_path()`, or `earwax.play_and_destroy()`, then a random file will be selected from the resulting directory.

The BufferDirectory class takes things one step further:

```
lasers: BufferDirectory = BufferDirectory(Path('sounds/weapons/lasers'))

laser_buffer: Buffer = lasers.random_buffer()
```

This will get you a random buffer from `lasers.buffers`.

Sometimes you may have other files in a sounds directory in addition to the sound files themselves, attribution information for example. If this is the case, simply pass a glob argument when instantiating the class, like so:

```
bd: BufferDirectory = BufferDirectory(Path('sounds/music'), glob='*.ogg')
```

In closing, the BufferDirectory class is useful if you have a directory of sound files, that you'll want at some point throughout the lifecycle of your game. Folders of music tracks, footstep sounds, and weapon sounds are just some of the examples that spring to mind.

8.8 Promises

Promises are a way of running different kinds of tasks with Earwax.

The term is shamelessly stolen from JavaScript, and Earwax's interpretation is largely the same: A promise is instantiated, and set to run. At some point in the future, the promise will have a value, which can be listened for with the `on_done()` event.

This part of the tutorial contains some further thoughts on using the different types of promise Earwax has to offer.

8.8.1 Threaded Promises

The inspiration for the `earwax.ThreadedPromise` class came from a game i was writing. I wanted to load assets, as well as data from the internet, and it was taking ages. While things were loading, the game appeared to crash, which obviously wasn't good.

With the `ThreadedPromise` class, you can leave something to work in another thread, while the main thread remains free to process input ETC. You can use the `on_done()` event to be notified of (and provided with) the return value from your function.

For example:

```
promise: ThreadedPromise = ThreadedPromise(game.thread_pool)


@promise.register_func
def long_running_task() -> str:
    # Something which takes forever...
    return 'Finished.'


@promise.event
```

(continues on next page)

(continued from previous page)

```
def on_done(value: str) -> None:
    game.output('Task complete.')


promise.run()
```

As you can see from the above code, you use the `register_func()` method to register the function to use. That function will be automatically called in another thread, and the result send to the `on_done()` event.

If your code is likely to raise an error, there is a `on_error()` event too:

```
from pyglet.event import event_handled


@promise.event
def on_error(e: Exception) -> bool:
    game.output('Error: %r.' % e)
    return event_handled
```

By default, the `on_error` event raises the passed error, so it is necessary to return the `event_handled` value to prevent any other handlers from firing.

For the sake of completeness, there is a `on_finally()` event too:

```
@promise.event
def on_finally() -> None:
    game.output('Done.')
```

This event will be dispatched when the promise has been completed, whether or not an exception was raised.

If you want to cancel, there is a `cancel()` method to do it with, and of course a `on_cancel()` event which will be dispatched.

It is unlikely that the actual function will be cancelled, but you can rest assured that no further events will be dispatched.

When you have created all of your events, you should use the `run()` method to start your promise running.

It is worth noting that although this particular part of the tutorial concerns the `ThreadedPromise` class, all of the events that have been mentioned are actually present on the `earwax.Promise` class, and it is simply up to subclasses to implement them.

8.8.2 Staggered Promises

The `earwax.StaggeredPromise` class, which should have probably been called the `ContinuationPromise` class, was created out of my desire to write MOO-style suspends in Python.

Using the class, you can simply yield a number, and your function will suspend for *approximately* that long:

```
from earwax.types import StaggeredPromiseGeneratorType

@StaggeredPromise.decorate
def promise() -> StaggeredPromiseGeneratorType:
    game.output('Starting now.')
    yield 2.0
    game.output('Still working.')
    yield 5.0
    game.output('Done.')
```

(continues on next page)

(continued from previous page)

```
promise.run()
```

The only event which differs from those found on :*Threaded Promises*, is the `on_next()` event.

This event is dispatched every time your promise function yields:

```
@promise.event
def on_next(delay: float) -> None:
print('Delay: %.2f' % delay)
```

8.9 Stories

Stories are a way to create simple games using Earwax with no code. Stories consist of rooms, which contain exits and objects. Objects and exits in turn have actions which can be performed on them.

This document attempts to layout the steps involved in creating and editing a story.

8.9.1 Prerequisites

Before getting started, let's make sure everything is installed correctly. This assumes you are comfortable with whatever terminal is offered by your system.

Make sure earwax is installed:

```
pip install -U earwax
```

Earwax is frequently changing, so it's important you have the latest version.

If you want to copy and paste with earwax, you'll need the Pyperclip package. Let's install that now:

```
pip install -U pyperclip
```

This package is not necessary, but when you're copy and pasting long sound paths, it's certainly helpful.

8.9.2 Getting Started

Before we can edit a story, we must first create one. To do this, we use the `story new` subcommand of earwax:

```
earwax story new world.yaml
```

You should see something like the following:

```
Created Untitled World.
```

The filename can be whatever you want, and you are free to rename or move this file as you wish. Be aware however, that unless the paths to the sound files you use are absolute, moving the file will not work as you expect.

8.9.3 Playing a Story

Stories can be played with the `story play` command, like so:

```
earwax story play world.yaml
```

You can replace `world.yaml` in the command above to be whatever filename you have chosen for your world.

8.9.4 Editing a Story

Now we have created a story, let's edit it.

When editing stories, you see the same interface as if you were a normal player. There are extra hotkeys of course, and the main menu changes to present you with extra options for configuring the over all story, as well as Earwax itself.

To get started, type:

```
earwax story edit world.yaml
```

The filename in that command should be the same one you gave to the `story new` command.

You will see a couple of log lines printed to your terminal's standard output, then you'll be in the main menu.

8.9.5 The Main Menu

The main menu is largely the same whether you're playing or editing a story. The difference is the number of items which are displayed.

Start new game

Takes you into the game world, where you can perform your edits.

This option is also present when playing a story.

Load game

Start with a loaded saved game.

This option is also present when playing a story.

Show warnings

This option will show you a list of any warnings which were generated while loading the story file.

When you first edit a game, there will be 1 warning. This is because the default room that is created has no exits leading from it.

Save story

This option will save any edits you have made so far. The story can also be saved by pressing control + s from within the story itself.

Configure Earwax

You can use this option to configure various parts of the game engine itself, such as the default menu sounds, and whether or not speech and braille are enabled.

When you have finished in this menu, you must activate the “Return to main menu” option at the end. This is so that the configuration can be saved, and you can be warned of any problems.

Add or remove credits

This option lets you add or remove credits from your game. This is useful if you plan to (or even need to) attribute someone for assets used in your story.

Set initial room

This option lets you set the room which the player will end up in when they first start playing your game.

It won’t always be the room they appear in when they start playing, because they can save their progress, and then load it using the `Load game` option.

Main menu music

This menu is where you can add or remove music from the main menu.

It is possible to have multiple tracks playing simultaneously, but you cannot alter their individual volumes.

World options

This menu allows you to rename your story, add an author, and set the default panning strategy.

Report Earwax bug

This option opens a web page where you can report a bug to Earwax.

As a personal note: Please please please use this if you find a problem. Letting me know personally is a great way to get your bug report lost.

Exit

This option is fairly self-explanatory: It quits the game and closes the window.

What it *doesn’t* do is save your work. You have to do that manually.

Credits

When you have added credits to your game, an option for viewing them will appear in the main menu.

This option won’t appear unless there are credits, since showing an empty credits menu to players would serve no purpose.

8.9.6 Start Game

Choosing the first option "Start new game", you will be placed into the first room.

Rooms

This room doesn't really have that much going for it: It's called "first_room", which incidentally is also its ID, and it has no meaningful description. Let's change that now.

Renaming Rooms

There are two ways to rename a room: With a new textual name, or by "shadowing" the name of another room.

Simple Renaming

You can rename anything with this first method. Press the `r` key on any object you want to rename, and you can type in a new name, before pressing enter.

Shadowing Names

Shadowing room names is only possible for rooms. It involves using the ID of another room, to "shadow" the name.

To do this, press `shift + r`. A menu will appear, showing every other room in the story. If you have no other rooms, this menu will be empty.

It is worth noting that shadowing room names and descriptions can only work for one level of rooms. That is, you cannot have room 1 shadow the name of room 2 which shadows the name of room 3. This is because you could also then have room 3 shadowing the name of room 1, which would cause an infinite loop.

Describing a Room

Rooms are the only things in stories which can be described. You can describe a room with the `e` key. The `d` key is not used, since this would conflict with dropping objects.

The key combination `shift + e` allows you to shadow the description of another room. Shadowing descriptions follows the same rules as shadowing names.

Adding New Rooms

A world wouldn't be much with only one room to visit. The way to create rooms - and incidentally exits and objects - is with the `c` key.

If you press the `c` key, a menu will appear, allowing you to select what you would like to create.

Selecting `Room` from the bottom of this menu, will create - and move you to - another empty room.

Moving Between Rooms

While exits are the primary way for *players* to move between rooms, it is helpful to have a quicker way as a builder.

Pressing the `g` key brings up a menu of rooms you can use to move quickly between rooms. This obviously bypasses exits, allowing you to get to as yet unlinked rooms.

Exits

Exits are the only way for *players* to move between rooms. They must be built to link rooms, otherwise there will be no way to access them.

Incidentally, unlinked (or inaccessible) rooms will result in warnings when editing worlds.

Building Exits

To create an exit, again use the `c` (create) key, and select `Exit`.

This will bring up a list of rooms (excluding the current one), which - when selected - will construct the exit.

Renaming Exits

You can rename an exit by first selecting it from the exits list, and pressing the `r` key.

Objects

The second entry in the create menu is for creating objects. You *must* be in the room where you plan to place the object before you create. Taking the object and dropping it elsewhere will not actually "move" the object, and currently there is no way to relocate objects.

This can be looked at if someone is upset by this lack enough to submit an issue.

Renaming Objects

You can rename an object by selecting it from the objects list, and pressing the `r` key.

Object Types

objects can have one of a couple of different types. You can change the object type with the `t` key.

The object types are listed below:

Cannot Be Taken

This type is best for stationary objects like scenery. It will not be possible to take such objects.

Can Be Taken

Objects of this type can be picked up. Their `take action` dictates what message and sound is presented to the player when they are taken.

If an object's `take action` is not set, the world's `take action` will be used instead.

Objects of this type cannot be dropped. If you think that's stupid, read on (there is another type).

Can Be Dropped

Objects of this type can both be picked up and dropped.

The object's `drop action` will be used to provide a message and a sound for when the object is dropped.

If there is no `drop action` on the object in question, the world's default `drop action` will be used instead.

Can Be Used

This final type is `not` listed in the types menu. It is only applicable when a `use action` is specified for an object. Otherwise, the object is considered unusable.

It is perfectly possible for an object to be usable but not droppable. It is even possible for an object to be usable, but impossible for that object to be picked up in the first place. Note that this would be pointless, since the `use action` can only be accessed by the player when the object is in their inventory.

Object Classes

Objects can belong to 0 or more `classes`. These classes are useful for grouping objects, and will be used to make exits allow or disallow player access in the future.

To keep apprised of the work on exits, please track this issue.

To add and remove classes from an object, use the `o` key.

Object classes can be added and removed with the key combination `shift + o`.

Messages

Objects, exits, and the world itself all have messages. To set messages, use the `m` key.

This key will set different messages depending on which category is shown:

- When in the `location` category, edit the world messages.
- When an entry from the `objects` category is selected, you can set the message that is shown when any object action is used.
- When an entry from the `exits` category is selected, you can set the message which is shown when using that exit.

Sounds

You can set sounds for objects and exits, as well as the world itself.

To set sounds, use the `s` key. This key performs different actions, depending on which category is shown:

- When in the `location` category, edit the world sounds.
- When an entry from the `objects` category is selected, you can set the sound which is heard when any object action is used.
- When an entry from the `exits` category is selected, you can set the sound which is heard when using that exit.

Ambiances

Using the `a` key, you can edit ambiances for the current room, and for objects.

Exits do *not* have ambiances, so the `a` key does nothing when in the `exits` category.

Actions

Actions are used throughout stories. They can be edited with the `shift + a` shortcut.

- When in the `location` category, you can edit (or clear) the default actions for the world.
- When an entry from the `objects` category is selected, you can edit (or delete) actions for when an object is taken, dropped, or used, or you can edit the custom actions for the given object.
- When an entry from the `exits` category is selected, you can edit (or clear) the action which is used when the exit is traversed.

8.9.7 Saving Stories

As mentioned in the *Save Story* section, you can save your story at any time with the keyboard shortcut `control + s`.

8.10 Building Stories

You can build your story into a Python file with the `story build` command.

Assuming you have a world file named `world.yaml`, you can convert it to python with the command:

```
earwax story build world.yaml world.py
```

This will output `world.py`. You can then play your story with:

```
python world.py
```

If you wish to consolidate all your sounds, you can use the `-s` switch:

```
earwax story build world.yaml world.py -s assets
```

This will copy all your sound files into a folder named `assets`. Their names will be changed, and the folder structure will be defined by earwax.

A note for screen reader users: It is not recommended that you read the generated python file line-by-line. This is because the line which holds the YAML data for your world can be extremely long, and this negatively impacts screen reader use.

CHAPTER 9

earwax

9.1 earwax package

9.1.1 Subpackages

earwax.cmd package

Subpackages

earwax.cmd.subcommands package

Submodules

earwax.cmd.subcommands.configure_earwax module

Provides the configure_earwax subcommand.

`earwax.cmd.subcommands.configure_earwax.`**`configure_earwax`**`(`*args: argparse.Namespace*`)` → None

Configure earwax, using a `earwax.ConfigMenu` instance.

earwax.cmd.subcommands.game module

Provides the game subcommand.

`earwax.cmd.subcommands.game.`**`new_game`**`(`*args: argparse.Namespace*`)` → None

Create a default game.

earwax.cmd.subcommands.game_map module

Provides subcommands for working with maps.

`earwax.cmd.subcommands.game_map.`**`edit_map`**`(`*args: argparse.Namespace*`)` → None
  Edit the map at the given filename.

`earwax.cmd.subcommands.game_map.`**`new_map`**`(`*args: argparse.Namespace*`)` → None
  Create a new map.

earwax.cmd.subcommands.init_project module

Provides the init_project subcommand.

`earwax.cmd.subcommands.init_project.`**`init_project`**`(`*args: argparse.Namespace*`)` → None
  Initialise or update the project at the given directory.

`earwax.cmd.subcommands.init_project.`**`update`**`()` → None
  Update the given path to conform to the latest earwax file structure.

  Parameters **`p`** – The path to update.

earwax.cmd.subcommands.story module

Provides the story subcommand.

`earwax.cmd.subcommands.story.`**`build_story`**`(`*args: argparse.Namespace*`)` → None
  Build the world.

`earwax.cmd.subcommands.story.`**`copy_action`**`(`*action: earwax.story.world.WorldAction*, *destination: pathlib.Path*, *index: int*`)` → None
  Copy the sound for the given action.

  Parameters

  - **`action`** – The action whose sound will be copied.
  - **`destination`** – The destination the sound will be copied to.

    If this directory does not exist, it will be created before the copy.
  - **`index`** – The number to base the resulting file name on.

`earwax.cmd.subcommands.story.`**`copy_actions`**`(`*actions: List[earwax.story.world.WorldAction], destination: pathlib.Path*`)` → None
  Copy the sounds from a list of action objects.

  Parameters

  - **`actions`** – The list of actions whose sounds will be copied.
  - **`destination`** – The destination for the copied sounds.

    If this directory does not exist, it will be created before the copy.

`earwax.cmd.subcommands.story.`**`copy_ambiances`**`(`*ambiances: List[earwax.story.world.WorldAmbiance], destination: pathlib.Path*`)` → None
  Copy all ambiance files.

  Parameters

  - **`ambiances`** – The ambiances whose sounds will be copied.

- **`destination`** – The ambiances directory to copy into.

  If this directory does not exist, it will be created before copying begins.

`earwax.cmd.subcommands.story.`**`copy_path`**(*source: Union[str, pathlib.Path], destination: pathlib.Path*) → str

Copy the given file or folder to the given destination.

Parameters

- **`source`** – Where to copy from.
- **`destination`** – The destination for the new file.

`earwax.cmd.subcommands.story.`**`create_story`**(*args: argparse.Namespace*) → None

Create a new story.

`earwax.cmd.subcommands.story.`**`edit_story`**(*args: argparse.Namespace*) → None

Edit the given story.

`earwax.cmd.subcommands.story.`**`get_filename`**(*filename: str, index: int*) → str

Return a unique filename.

Given a filename of `'music/track.wav'`, and an index of `5`, `'5.wav'` would be returned.

Parameters

- **`filename`** – The original filename (can include path).
- **`index`** – The index of this filename in whatever list is being iterated over.

`earwax.cmd.subcommands.story.`**`make_directory`**(*directory: pathlib.Path*) → None

Make the given directory, if necessary.

if the given directory already exists, print a message to that effect.

Otherwise, create the directory, and print a message about it.

Parameters **`directory`** – The directory to create.

`earwax.cmd.subcommands.story.`**`play_story`**(*args: argparse.Namespace, edit: bool = False*) → None

Load and play a story.

earwax.cmd.subcommands.vault module

Provides subcommands for working with vault files.

`earwax.cmd.subcommands.vault.`**`compile_vault`**(*args: argparse.Namespace*) → None

Compile the given vault file.

`earwax.cmd.subcommands.vault.`**`new_vault`**(*args: argparse.Namespace*) → None

Create a new vault file.

Module contents

A directory containing sub commands for the earwax utility.

Submodules

earwax.cmd.constants module

Provides various constants used by the script.

earwax.cmd.game_level module

Provides the GameLevel class.

class `earwax.cmd.game_level.`**`BoxLevelData`**(*bearing: int = NOTHING*)

Bases: *`earwax.mixins.DumpLoadMixin`*

A box level.

An instance of this class can be used to build a `earwax.BoxLevel` instance.

class `earwax.cmd.game_level.`**`GameLevel`**(*name: str, data: Union[earwax.cmd.game_level.LevelData, earwax.cmd.game_level.BoxLevelData], scripts: List[earwax.cmd.game_level.GameLevelScript] = NOTHING, id: str = NOTHING*)

Bases: *`earwax.mixins.DumpLoadMixin`*

A game level.

This class is used in the GUI so that non-programmers can can create levels with no code.

Variables

- **`name`** – The name of this level.
- **`data`** – The data for this level.
- **`scripts`** – The scripts that are attached to this level.

class `earwax.cmd.game_level.`**`GameLevelScript`**(*name: str, trigger: earwax.cmd.game_level.Trigger, id: str = NOTHING*)

Bases: *`earwax.mixins.DumpLoadMixin`*

A script which is attached to a game level.

`code`

Return the code of this script.

If *`script_path`* does not exist, an empty string will be returned.

`script_name`

Return the script name (although not the path) for this script.

If you want the path, use the *`script_path`* attribute.

`script_path`

Return the path where code for this script resides.

If you want the filename, use the *`script_name`* attribute.

class `earwax.cmd.game_level.`**`LevelData`**

Bases: *`earwax.mixins.DumpLoadMixin`*

A standard earwax level.

An instance of this class can be used to build a `earwax.Level` instance.

class earwax.cmd.game_level.**Trigger**(*symbol: Optional[str] = None*, *modifiers: List[str] = NOTHING*, *mouse_button: Optional[str] = None*, *hat_directions: Optional[str] = None*, *joystick_button: Optional[int] = None*)

Bases: *earwax.mixins.DumpLoadMixin*

A trigger that can activate a function in a game.

earwax.cmd.keys module

Provides keys for templates.

earwax.cmd.main module

The Earwax command line utility.

This module provides the `cmd_main` function, and all sub commands.

To run the client:

- Make sure Earwax and all its dependencies are up to date.
- **In the folder where you wish to work, type `earwax`. This is a standard** command line utility, which should provide enough of its own help that no replication is required in this document.

NOTE: At the time of writing, only the `earwax story` command actually does all that much that is useful. Everything else needs fleshing out.

If you want to create more subcommands, add them in the subcommands directory, then register them with the *subcommand()* method.

earwax.cmd.main.**add_help**(*subparser: argparse._SubParsersAction*) → argparse.ArgumentParser

Add a help command to any subcommand.

earwax.cmd.main.**add_subcommands**(*_parser: argparse.ArgumentParser*) → argparse._SubParsersAction

Add subcommands to any parser.

Parameters **_parser** – The parser to add subcommands to.

earwax.cmd.main.**cmd_help**(*subcommand: argparse._SubParsersAction*) → Callable[[argparse.Namespace], None]

Return a command function that will show all subcommands.

earwax.cmd.main.**cmd_main**() → None

Run the earwax client.

earwax.cmd.main.**subcommand**(*name: str*, *func: Callable[[argparse.Namespace], None]*, *subparser: argparse._SubParsersAction*, *formatter_class: Type[argparse.HelpFormatter] = <class 'argparse.ArgumentDefaultsHelpFormatter'>*, ***kwargs*) → argparse.ArgumentParser

Add a subcommand to the argument parser.

Parameters

- **name** – The name of the new command.
- **func** – The function that will be called when this subcommand is used.
- **subparser** – The parser to add the sub command to.

- **kwargs** – Keyword arguments to be passed to `commands.add_parser`.

earwax.cmd.project module

Provides the Workspace class.

class `earwax.cmd.project.`**Project**(*name: str, author: str = NOTHING, description: str = NOTHING, version: str = NOTHING, requirements: str = NOTHING, credits: List[earwax.cmd.project_credit.ProjectCredit] = NOTHING, variables: List[earwax.cmd.variable.Variable] = NOTHING, levels: List[earwax.cmd.game_level.GameLevel] = NOTHING*)

Bases: `earwax.mixins.DumpLoadMixin`

An earwax project.

This object holds the id of the initial map (if any), as well as global variables the user can create with the `global` subcommand.

Variables

- **name** – The name of this project.
- **author** – The author of this project.
- **description** – A description for this project.
- **version** – The version string of this project.
- **initial_map_id** – The id of the first map to load with the game.
- **credits** – A list of credits for this project.
- **variables** – The variables created for this project.

earwax.cmd.project_credit module

Provides the ProjectCredit class.

class `earwax.cmd.project_credit.`**ProjectCredit**(*name: str, url: str, sound: Optional[str], loop: bool*)

Bases: `earwax.mixins.DumpLoadMixin`

A representation of the `earwax.Credit` class.

This class has a different name to avoid possible confusion.

Variables

- **name** – The name of what is being credited.
- **url** – A URL for this credit.
- **sound** – The sound that will play when this credit is shown in a menu.
- **loop** – Whether or not `ProjectCredit.sound` should loop.

path

Return `ProjectCredit.sound` as a path.

earwax.cmd.variable module

Provides the Variable class.

class earwax.cmd.variable.**Variable**(*name: str*, *type: earwax.cmd.variable.VariableTypes*, *value: T*, *id: str = NOTHING*)

Bases: `typing.Generic`, *earwax.mixins.DumpLoadMixin*

A variable in a game made with the earwax script.

Variables

- **name** – The name of the variable.
- **type** – The type of `value`.
- **value** – The value this variable holds.
- **id** – The id of this variable.

get_type() → earwax.cmd.variable.VariableTypes

Return the type of this variable.

This method returns a member of *VariableTypes*.

classmethod load(*data: Dict[str, Any]*, **args*) → earwax.cmd.variable.Variable

Load a variable, and check its type.

Parameters **value** – The value to load.

class earwax.cmd.variable.**VariableTypes**

Bases: `enum.Enum`

Provides the possible types of variable.

Variables

- ***type_int*** – An integer.
- ***type_float*** – A floating point number.
- ***type_string*** – a string.
- ***type_bool*** – A boolean value.

type_bool = 3

type_float = 1

type_int = 0

type_string = 2

Module contents

Earwax Script.

Command Line

This program allows you to create games with very little actual coding.

This document will be updated as this program matures.

`earwax.cmd.`**`cmd_main`**`()` → None

Run the earwax client.

earwax.mapping package

Submodules

earwax.mapping.box module

Provides box-related classes, functions, and exceptions.

`class` `earwax.mapping.box.`**`Box`**`(`*game: Game, start: earwax.point.Point, end: earwax.point.Point, name: Optional[str] = None, surface_sound: Optional[pathlib.Path] = None, wall_sound: Optional[pathlib.Path] = None, type: earwax.mapping.box.BoxTypes = NOTHING, data: Optional[T] = None, stationary: bool = NOTHING, reverb: Optional[object] = NOTHING, box_level: Optional[BoxLevel] = None*`)`

Bases: `typing.Generic`, *`earwax.mixins.RegisterEventMixin`*

A box on a map.

You can create instances of this class either singly, or by using the `earwax.Box.create_row()` method.

If you already have a list of boxes, you can fit them all onto one map with the `earwax.Box.create_fitted()` method.

Boxes can be assigned arbitrary user data:

```
b: Box[Enemy] = Box(start, end, data=Enemy())
b.enemy.do_something()
```

In addition to the coordinates supplied to this class's constructor, a `earwax.BoxBounds` instance is created as `earwax.Box.bounds`.

This class uses the pyglet.event framework, so you can register and dispatch events in the same way you would with `pyglet.window.Window`, or any other `EventDispatcher` subclass.

Variables

- **`game`** – The game that this box will work with.
- **`start`** – The coordinates at the bottom rear left corner of this box.
- **`end`** – The coordinates at the top front right corner of this box.
- **`name`** – An optional name for this box.
- **`surface_sound`** – The sound that should be heard when walking in this box.
- **`wall_sound`** – The sound that should be heard when colliding with walls in this box.
- **`type`** – The type of this box.
- **`data`** – Arbitrary data for this box.
- **`bounds`** – The bounds of this box.
- **`centre`** – The point that lies at the centre of this box.
- **`reverb`** – The reverb that is assigned to this box.

close() → None

Close the attached door.

If this box is a door, set the `open` attribute of its `data` to `False`, and play the appropriate sound. Otherwise, raise `earwax.NotADoor`.

Parameters **door** – The door to close.

contains_point(*coordinates: earwax.point.Point*) → bool

Return whether or not this box contains the given point.

Returns `True` if this box spans the given coordinates, `False` otherwise.

Parameters **coordinates** – The coordinates to check.

could_fit(*box: earwax.mapping.box.Box*) → bool

Return whether or not the given box could be contained by this one.

Returns `True` if the given box could be contained by this box, `False` otherwise.

This method behaves like the `contains_point()` method, except that it works with `Box` instances, rather than `Point` instances.

This method doesn't care about the `parent` attribute on the given box. This method simply checks that the `start` and `end` points would fit inside this box.

Parameters **box** – The box whose bounds will be checked.

classmethod create_fitted(*game: Game, children: List[Box], pad_start: Optional[earwax.point.Point] = None, pad_end: Optional[earwax.point.Point] = None, **kwargs*) → Box

Return a box that fits all of `children` inside itself.

Pass a list of `Box` instances, and you'll get a box with its `start`, and `end` attributes set to match the outer bounds of the provided children.

You can use `pad_start`, and `pad_end` to add or subtract from the calculated start and end coordinates.

Parameters

- **children** – The list of `Box` instances to encapsulate.
- **pad_start** – A point to add to the calculated start coordinates.
- **pad_end** – A point to add to the calculated end coordinates.
- **kwargs** – The extra keyword arguments to pass to `Box.__init__`.

classmethod create_row(*game: Game, start: earwax.point.Point, size: earwax.point.Point, count: int, offset: earwax.point.Point, get_name: Optional[Callable[[int], str]] = None, on_create: Optional[Callable[[Box], None]] = None, **kwargs*) → List[Box]

Generate a list of boxes.

This method is useful for creating rows of buildings, or rooms on a corridor to name a couple of examples.

It can be used like so:

```
offices = Box.create_row(
    game,  # Every Box instance needs a game.
    Point(0, 0),  # The bottom_left corner of the first box.
    Point(3, 2, 0),  # The size of each box.
    3,  # The number of boxes to build.
    # The next argument is how far to move from the top right
    # corner of each created box:
```

(continues on next page)

(continued from previous page)

```
    Point(1, 0, 0),
    # We want to name each room. For that, there is a function!
    get_name=lambda i: f'Room {i + 1}',
    # Let's make them all rooms.
    type=RoomTypes.room
)
```

This will result in a list containing 3 rooms:

- The first from (0, 0, 0) to (2, 1, 0)
- The second from (3, 0, 0) to (5, 1, 0)
- And the third from (6, 0, 0) to (8, 1, 0)

PLEASE NOTE: If none of the size coordinates are `>= 1`, the top right coordinate will be less than the bottom left, so `get_containing_box()` won't ever find it.

Parameters

- **`start`** – The `start` coordinate of the first box.
- **`size`** – The size of each box.
- **`count`** – The number of boxes to build.
- **`offset`** – The distance between the boxes.

  If no coordinate of the given value is `>= 1`, overlaps will occur.
- **`get_name`** – A function which should return an appropriate name.

  This function will be called with the current position in the loop.

  0 for the first room, 1 for the second, and so on.
- **`on_create`** – A function which will be called after each box is created.

  The only provided argument will be the box that was just created.
- **`kwargs`** – Extra keyword arguments to be passed to `Box.__init__`.

`get_nearest_point`(*point: earwax.point.Point*) → earwax.point.Point
Return the point on this box nearest to the provided point.

Parameters **`point`** – The point to start from.

`handle_door`() → None
Open or close the door attached to this box.

`handle_portal`() → None
Activate a portal attached to this box.

`is_door`
Return `True` if this box is a door.

`is_portal`
Return `True` if this box is a portal.

`is_wall`(*p: earwax.point.Point*) → bool
Return `True` if the provided point is inside a wall.

Parameters **`p`** – The point to interrogate.

on_activate() → None
Handle the enter key.

This event is dispatched when the player presses the enter key.

It is guaranteed that the instance this event is dispatched on is the one the player is stood on.

on_close() → None
Handle this box being closed.

on_collide(*coordinates: earwax.point.Point*) → None
Play an appropriate wall sound.

This function will be called by the Pyglet event framework, and should be called when a player collides with this box.

on_footstep(*bearing: float*, *coordinates: earwax.point.Point*) → None
Play an appropriate surface sound.

This function will be called by the Pyglet event framework, and should be called when a player is walking on this box.

This event is dispatched by `earwax.BoxLevel.move` upon a successful move.

Parameters **coordinates** – The coordinates the player has just moved to.

on_open() → None
Handle this box being opened.

open() → None
Open the attached door.

If this box is a door, set the `open` attribute of its `data` to `True`, and play the appropriate sound. Otherwise, raise `earwax.NotADoor`.

Parameters **box** – The box to open.

scheduled_close(*dt: float*) → None
Call `close()`.

This method will be called by `pyglet.clock.schedule_once`.

Parameters **dt** – The `dt` parameter expected by Pyglet's schedule functions.

sound_manager
Return a suitable sound manager.

class earwax.mapping.box.**BoxBounds**(*bottom_back_left: earwax.point.Point*, *top_front_right: earwax.point.Point*)
Bases: `object`

Bounds for a `earwax.Box` instance.

Variables

- **bottom_back_left** – The bottom back left point.
- **top_front_right** – The top front right point.
- **bottom_front_left** – The bottom front left point.
- **bottom_front_right** – The bottom front right point.
- **bottom_back_right** – The bottom back right point.
- **top_back_left** – The top back left point.
- **top_front_left** – The top front left point.

- **`top_back_right`** – The top back right point.

`area`
Return the area of the box.

`depth`
Get the depth of this box (front to back).

`height`
Return the height of this box.

`is_edge`(*p: earwax.point.Point*) → bool
Return `True` if `p` represents an edge.

Parameters **`p`** – The point to interrogate.

`volume`
Return the volume of this box.

`width`
Return the width of this box.

exception `earwax.mapping.box.`**`BoxError`**
Bases: `Exception`

General box level error.

class `earwax.mapping.box.`**`BoxTypes`**
Bases: `enum.Enum`

The type of a box.

Variables

- **`empty`** – Empty space.

  Boxes of this type can be traversed wit no barriers.

- **`room`** – An open room with walls around the edge.

  Boxes of this type can be entered by means of a door. The programmer must provide some means of exit.

- **`solid`** – Signifies a solid, impassible barrier.

  Boxes of this type cannot be traversed.

`empty = 0`

`room = 1`

`solid = 2`

exception `earwax.mapping.box.`**`NotADoor`**
Bases: *`earwax.mapping.box.BoxError`*

The current box is not a door.

exception `earwax.mapping.box.`**`NotAPortal`**
Bases: *`earwax.mapping.box.BoxError`*

The current box is not a portal.

earwax.mapping.box_level module

Provides the BoxLevel class.

class earwax.mapping.box_level.**BoxLevel**(*game: Game, boxes: List[earwax.mapping.box.Box[typing.Any][Any]] = NOTHING, coordinates: earwax.point.Point = NOTHING, bearing: int = 0, current_box: Optional[earwax.mapping.box_level.CurrentBox] = None*)

Bases: *earwax.level.Level*

A level that deals with sound generation for boxes.

This level can be used in your games. Simply bind the various action methods (listed below) to whatever triggers suit your purposes.

Some of the attributes of this class refer to a "perspective". This could theoretically be anything you want, but most likely refers to the player. Possible exceptions include if you made an instance to represent some kind of long range vision for the player.

Action-ready Methods

- `move()`.
- `show_coordinates()`
- `show_facing()`
- `turn()`
- `show_nearest_door()`
- `describe_current_box()`

Variables

- **box** – The box that this level will work with.
- **coordinates** – The coordinates of the perspective.
- **bearing** – The direction the perspective is facing.
- **current_box** – The most recently walked over box.

  If you don't set this attribute when creating the instance, then the first time the player moves using the `move()` method, the name of the box they are standing on will be spoken.
- **reverb** – An optional reverb to play sounds through.

  You shouldn't write to this property, instead use the `connect_reverb()` method to set a new reverb, and `disconnect_reverb()` to clear.

activate(*door_distance: float = 2.0*) → Callable[[], None]

Return a function that can be call when the enter key is pressed.

First we check if the current box is a portal. If it is, then we call `handle_portal()`.

If it is not, we check to see if there is a door close enough to be opened or closed. If there is, then we call `handle_door()` on it.

If none of this works, and there is a current box, dispatch the `on_activate()` event to let the box do its own thing.

Parameters **`door_distance`** – How close doors have to be for this method to open or close them.

`add_box`(*box: earwax.mapping.box.Box[typing.Any][Any]*) → None
Add a box to `self.boxes`.

Parameters **`box`** – The box to add.

`add_boxes`(*boxes: Iterable[earwax.mapping.box.Box]*) → None
Add multiple boxes with one call.

Parameters **`boxes`** – An iterable for boxes to add.

`add_default_actions`() → None
Add some default actions.

This method adds the following actions:

- Move forward: W
- Turn 180 degrees: S
- Turn 45 degrees left: A
- Turn 45 degrees right: D
- Show coordinates: C
- Show the facing direction: F
- Describe current box: X
- Speak nearest door: Z
- Activate nearby objects: Return

`calculate_coordinates`(*distance: float*, *bearing: int*) → Tuple[float, float]
Calculate coordinates at the given distance in the given direction.

Used by `move()` to calculate new coordinates.

Override this method if you want to change the algorithm used to calculate the target coordinates.

Please bear in mind however, that the coordinates this method returns should always be 2d.

Parameters

- **`distance`** – The distance which should be used.
- **`bearing`** – The bearing the new coordinates are in.

  This value may not be the same as `self.bearing`.

`collide`(*box: earwax.mapping.box.Box[typing.Any][Any], coordinates: earwax.point.Point*) → None
Handle collitions.

Called to run collision code on a box.

Parameters

- **`box`** – The box the player collided with.
- **`coordinates`** – The coordinates the player was trying to reach.

`describe_current_box`() → None
Describe the current box.

get_angle_between(*other: earwax.point.Point*) → float
Return the angle between the perspective and the other coordinates.

This function takes into account `self.bearing`.

Parameters **other** – The target coordinates.

get_boxes(*t: Any*) → List[earwax.mapping.box.Box]
Return a list of boxes of the current type.

If no boxes are found, an empty list is returned.

Parameters **t** – The type of the boxes.

get_containing_box(*coordinates: earwax.point.Point*) → Optional[earwax.mapping.box.Box]
Return the box that spans the given coordinates.

If no box is found, `None` will be returned.

This method scans `self.boxes` using the `sort_boxes()` method.

Parameters **coordinates** – The coordinates the box should span.

get_current_box() → Optional[earwax.mapping.box.Box]
Get the box that lies at the current coordinates.

handle_box(*box: earwax.mapping.box.Box[typing.Any][Any]*) → None
Handle a bulk standard box.

The coordinates have already been set, and the `on_footstep` event dispatched, so all that is left is to speak the name of the new box, if it is different to the last one, update `self.reverb` if necessary, and store the new box.

move(*distance: float = 1.0*, *vertical: Optional[float] = None*, *bearing: Optional[int] = None*) → Callable[[], None]
Return a callable that allows the player to move on the map.

If the move is successful (I.E.: There is a box at the destination coordinates), the `on_move()` event is dispatched.

If not, then `on_move_fail()` is dispatched.

Parameters

- **distance** – The distance to move.
- **vertical** – An optional adjustment to be added to the vertical position.
- **bearing** – An optional direction to move in.

  If this value is `None`, then `self.bearing` will be used.

nearest_by_type(*start: earwax.point.Point*, *data_type: Any*, *same_z: bool = True*) → Optional[earwax.mapping.box_level.NearestBox]
Get the nearest box to the given point by type.

If no boxes of the given type are found, `None` will be returned.

Parameters

- **start** – The point to start looking from.
- **data_type** – The type of `box data` to search for.
- **same_z** – If this value is `True`, only boxes on the same z axis will be considered.

nearest_door(*start: earwax.point.Point*, *same_z: bool = True*) → Optional[earwax.mapping.box_level.NearestBox]

Get the nearest door.

Iterates over all doors, and returned the nearest one.

Parameters

- **start** – The coordinates to start from.
- **same_z** – If `True`, then doors on different levels will not be considered.

nearest_portal(*start: earwax.point.Point*, *same_z: bool = True*) → Optional[earwax.mapping.box_level.NearestBox]

Return the nearest portal.

Parameters

- **start** – The coordinates to start from.
- **same_z** – If `True`, then portals on different levels will not be considered.

on_move_fail(*distance: float, vertical: Optional[float], bearing: int, coordinates: earwax.point.Point*) → None

Handle a move failure.

An event that will be dispatched when the `move()` action has been used, but no move was performed.

Parameters

- **distance** – The `distance` value that was passed to `move()`.
- **vertical** – The `vertical` value that was passed to `move`.
- **bearing** – The `bearing` argument that was passed to `move`, or `self.bearing`.

on_move_success() → None

Handle a successful move.

An event that will be dispatched when the `move()` action is used.

By default, this method plays the correct footstep sound.

on_push() → None

Set listener orientation, and start ambiances and tracks.

on_turn() → None

Handle turning.

An event that will dispatched when the `turn()` action is used.

register_box(*box: earwax.mapping.box.Box*) → None

Register a box that is already in the boxes list.

Parameters box – The box to register.

remove_box(*box: earwax.mapping.box.Box[typing.Any][Any]*) → None

Remove a box from `self.boxes`.

Parameters box – The box to remove.

set_bearing(*angle: int*) → None

Set the direction of travel and the listener's orientation.

Parameters angle – The bearing (in degrees).

set_coordinates(*p: earwax.point.Point*) → None
Set the current coordinates.

Also set listener position.

Parameters p – The new point to assign to `self.coordinates`.

show_coordinates(*include_z: bool = False*) → Callable[[], None]
Speak the current coordinates.

show_facing(*include_angle: bool = True*) → Callable[[], None]
Return a function that will let you see the current bearing as text.

For example:

```
l = BoxLevel(...)
l.action('Show facing', symbol=key.F)(l.show_facing())
```

Parameters include_angle – If `True`, then the actual angle will be shown along with the direction name.

show_nearest_door(*max_distance: Optional[float] = None*) → Callable[[], None]
Return a callable that will speak the position of the nearest door.

Parameters max_distance – The maximum distance between the current coordinates and the nearest door where the door will still be reported.

If this value is `None`, then any door will be reported.

sort_boxes() → List[earwax.mapping.box.Box]
Return `children` sorted by area.

turn(*amount: int*) → Callable[[], None]
Return a turn function.

Return a function that will turn the perspective by the given amount and dispatch the `on_turn` event.

For example:

```
l = BoxLevel(...)
l.action('Turn right', symbol=key.D)(l.turn(45))
l.action('Turn left', symbol=key.A)(l.turn(-45))
```

The resulting angle will always be in the range 0-359.

Parameters amount – The amount to turn by.

Positive numbers turn clockwise, while negative numbers turn anticlockwise.

class earwax.mapping.box_level.**CurrentBox**(*coordinates: earwax.point.Point, box: earwax.mapping.box.Box[typing.Any][Any]*)
Bases: `object`

Store a reference to the current box.

This class stores the position too, so that caching can be performed.

Variables

- **coordinates** – The coordinates that were last checked.
- **box** – The last current box.

class earwax.mapping.box_level.**NearestBox**(*box: earwax.mapping.box.Box*, *coordinates: earwax.point.Point*, *distance: float*)

Bases: `object`

A reference to the nearest box.

Variables

- **box** – The box that was found.
- **coordinates** – The nearest coordinates to the ones specified.
- **distance** – The distance between the supplied coordinates, and `coordinates`.

earwax.mapping.door module

Provides the Door class.

class earwax.mapping.door.**Door**(*open: bool = True*, *closed_sound: Optional[pathlib.Path] = None*, *open_sound: Optional[pathlib.Path] = None*, *close_sound: Optional[pathlib.Path] = None*, *close_after: Union[float, Tuple[float, float], None] = None*, *can_open: Optional[Callable[[], bool]] = None*, *can_close: Optional[Callable[[], bool]] = None*)

Bases: `object`

An object that can be added to a box to optionally block travel.

Doors can currently either be open or closed. When opened, they can optionally close after a specified time:

```
Door()  # Standard open door.
Door(open=False)  # Closed door.
Door(close_after=5.0)  # Will automatically close after 5 seconds.
# A door that will automatically close between 5 and 10 seconds after
# it has been opened:
Door(close_after=(5.0, 10.0)
```

Variables

- **open** – Whether or not this box can be walked on.

  If this value is `False`, then the player will hear `closed_sound` when trying to walk on this box.

  If this value is `True`, the player will be able to enter the box as normal.
- **closed_sound** – The sound that will be heard if `open` is `False`.
- **open_sound** – The sound that will be heard when opening this door.
- **close_sound** – The sound that will be heard when closing this door.
- **close_after** – When (if ever) to close the door after it has been opened.

  This attribute supports 3 possible values:

  – `None`: The door will not close on its own.

  – **A tuple of two positive floats a and b: A random number** between `a` and `b` will be selected, and the door will automatically close after that time.

  – A float: The exact time the door will automatically close after.

- **`can_open`** – An optional method which will be used to decide whether or not this door can be opened at this time.

  This method must return `True` or `False`, and must handle any messages which should be sent to the player.

- **`can_close`** – An optional method which will be used to decide whether or not this door can be closed at this time.

  This method must return `True` or `False`, and must handle any messages which should be sent to the player.

earwax.mapping.map_editor module

Provides the MapEditor class.

class `earwax.mapping.map_editor.`**`AnchorPoints`**

Bases: `enum.Enum`

The corners of a box points can be anchored to.

`bottom_back_left = 0`

`bottom_back_right = 4`

`bottom_front_left = 2`

`bottom_front_right = 3`

`top_back_left = 5`

`top_back_right = 7`

`top_front_left = 6`

`top_front_right = 1`

class `earwax.mapping.map_editor.`**`BoxPoint`**(*box_id: Optional[str] = None*, *corner: Optional[earwax.mapping.map_editor.AnchorPoints] = None*, *x: int = 0*, *y: int = 0*, *z: int = 0*)

Bases: *`earwax.mixins.DumpLoadMixin`*

Anchor a point to another box.

class `earwax.mapping.map_editor.`**`BoxTemplate`**(*start: earwax.mapping.map_editor.BoxPoint = NOTHING*, *end: earwax.mapping.map_editor.BoxPoint = NOTHING*, *name: str = 'Untitled Box'*, *surface_sound: Optional[str] = None*, *wall_sound: Optional[str] = None*, *type: earwax.mapping.box.BoxTypes = NOTHING*, *id: str = NOTHING*, *label: str = NOTHING*)

Bases: *`earwax.mixins.DumpLoadMixin`*

A template for creating a box.

Instances of this class will be dumped to the map file.

`get_default_label`() → str

Get a unique ID.

exception `earwax.mapping.map_editor.`**`InvalidLabel`**

Bases: `Exception`

An invalid ID or label was given.

class `earwax.mapping.map_editor.`**`LevelMap`**(*box_templates: List[earwax.mapping.map_editor.BoxTemplate] = NOTHING, coordinates: earwax.mapping.map_editor.BoxPoint = NOTHING, bearing: int = 0, name: str = 'Untitled Map', notes: str = NOTHING*)

Bases: *`earwax.mixins.DumpLoadMixin`*

A representation of a `earwax.BoxLevel` instance.

class `earwax.mapping.map_editor.`**`MapEditor`**(*game: Game, boxes: List[earwax.mapping.box.Box[typing.Any][Any]] = NOTHING, coordinates: earwax.point.Point = NOTHING, bearing: int = 0, current_box: Optional[earwax.mapping.box_level.CurrentBox] = None, filename: Optional[pathlib.Path] = None, context: earwax.mapping.map_editor.MapEditorContext = NOTHING*)

Bases: *`earwax.mapping.box_level.BoxLevel`*

A level which can be used for editing maps.

When this level talks about a map, it talks about a *`earwax.mapping.map_editor.LevelMap`* instance.

`box_menu`(*box: earwax.mapping.map_editor.MapEditorBox*) → None

Push a menu to configure the provided box.

`box_sound`(*template: earwax.mapping.map_editor.BoxTemplate, name: str*) → Callable[[], Generator[None, None, None]]

Push an editor for setting the given sound.

Parameters

- **`template`** – The template to modify.
- **`name`** – The name of the sound to modify.

`box_sounds`() → None

Push a menu for configuring sounds.

`boxes_menu`() → None

Push a menu to select a box to configure.

If there is only 1 box, it will not be shown.

`complain_box`() → None

Complain about there being no box.

`create_box`() → None

Create a box, then call `box_menu()`.

`get_default_context`() → earwax.mapping.map_editor.MapEditorContext

Return a suitable context.

`id_box`() → Generator[None, None, None]

Change the ID for the current box.

label_box() → Generator[None, None, None]
Rename the current box.

on_move_fail(*distance: float, vertical: Optional[float], bearing: int, coordinates: earwax.point.Point*) → None
Tell the user their move failed.

point_menu(*template: earwax.mapping.map_editor.BoxTemplate, point: earwax.mapping.map_editor.BoxPoint*) → Callable[[], None]
Push a menu for configuring individual points.

points_menu() → None
Push a menu for moving the current box.

rename_box() → Generator[None, None, None]
Rename the current box.

save() → None
Save the map level.

class `earwax.mapping.map_editor.`**MapEditorBox**(*game: Game, start: earwax.point.Point, end: earwax.point.Point, name: Optional[str] = None, surface_sound: Optional[pathlib.Path] = None, wall_sound: Optional[pathlib.Path] = None, type: earwax.mapping.box.BoxTypes = NOTHING, data: Optional[T] = None, stationary: bool = NOTHING, reverb: Optional[object] = NOTHING, box_level: Optional[BoxLevel] = None, id: str = NOTHING*)

Bases: `earwax.mapping.box.Box`

A box with an ID.

get_default_id() → str
Raise an error if the id is not provided.

class `earwax.mapping.map_editor.`**MapEditorContext**(*level: MapEditor, level_map: earwax.mapping.map_editor.LevelMap, template_ids: Dict[str, earwax.mapping.map_editor.BoxTemplate] = NOTHING, box_ids: Dict[str, earwax.mapping.box.Box[str][str]] = NOTHING*)

Bases: `object`

A context to hold map information.

This class acts as an interface between a `LevelMap` instance, and a `MapEditor` instance.

add_template(*template: earwax.mapping.map_editor.BoxTemplate, box: Optional[earwax.mapping.map_editor.MapEditorBox] = None*) → None
Add a template to this context.

This method will add the given template to its `box_template_ids` dictionary.

Parameters **template** – The template to add.

reload_template(*template: earwax.mapping.map_editor.BoxTemplate*) → None
Reload the given template.

This method recreates the box associated with the given template.

Parameters **`template`** – The template to reload.

`to_box`(*template: earwax.mapping.map_editor.BoxTemplate*) → earwax.mapping.map_editor.MapEditorBox
Return a box from a template.

Parameters **`template`** – The template to convert.

`to_point`(*data: earwax.mapping.map_editor.BoxPoint*) → earwax.point.Point
Return a point from the given data.

Parameters **`data`** – The `BoxPoint` to load the point from.

`earwax.mapping.map_editor.`**`iskeyword`**`()`
x.__contains__(y) <==> y in x.

`earwax.mapping.map_editor.`**`valid_label`**(*text: str*) → None
Ensure the given label or ID is valid.

If it could not be used as a Python identifier for any reason, *`earwax.mapping.map_editor.InvalidLabel`* will be raised.

Parameters **`text`** – The text to check.

earwax.mapping.portal module

Provides the Portal class.

`class` `earwax.mapping.portal.`**`Portal`**(*level: BoxLevel, coordinates: earwax.point.Point, bearing: Optional[int] = None, enter_sound: Optional[pathlib.Path] = None, exit_sound: Optional[pathlib.Path] = None, can_use: Optional[Callable[[], bool]] = None*)
Bases: *`earwax.mixins.RegisterEventMixin`*

A portal to another map.

An object that can be added to a `earwax.Box` to make a link between two maps.

This class implements `pyglet.event.EventDispatcher`, so events can be registered and dispatched on it.

The currently-registered events are:

- `on_enter()`
- `on_exit()`

Variables

- **`level`** – The destination level.
- **`coordinates`** – The exit coordinates.
- **`bearing`** – If this value is `None`, then it will be used for the player's bearing after this portal is used. Otherwise, the bearing from the old level will be used.
- **`enter_sound`** – The sound that should play when entering this portal.

  This sound is probably only used when an NPC uses the portal.
- **`exit_sound`** – The sound that should play when exiting this portal.

  This is the sound that the player will hear when using the portal.

- **can_use** – An optional method which will be called to ensure that this portal can be used at this time.

  This function should return `True` or `False`, and should handle any messages which should be sent to the player.

on_enter() → None
Handle a player entering this portal.

on_exit() → None
Handle a player exiting this portal.

Module contents

Mapping functions and classes for Earwax.

This module is inspired by Camlorn's post at this link.

All credit goes to him for the idea.

class earwax.mapping.**Box**(*game: Game*, *start: earwax.point.Point*, *end: earwax.point.Point*, *name: Optional[str] = None*, *surface_sound: Optional[pathlib.Path] = None*, *wall_sound: Optional[pathlib.Path] = None*, *type: earwax.mapping.box.BoxTypes = NOTHING*, *data: Optional[T] = None*, *stationary: bool = NOTHING*, *reverb: Optional[object] = NOTHING*, *box_level: Optional[BoxLevel] = None*)
Bases: `typing.Generic`, *`earwax.mixins.RegisterEventMixin`*

A box on a map.

You can create instances of this class either singly, or by using the `earwax.Box.create_row()` method.

If you already have a list of boxes, you can fit them all onto one map with the `earwax.Box.create_fitted()` method.

Boxes can be assigned arbitrary user data:

```
b: Box[Enemy] = Box(start, end, data=Enemy())
b.enemy.do_something()
```

In addition to the coordinates supplied to this class's constructor, a `earwax.BoxBounds` instance is created as `earwax.Box.bounds`.

This class uses the pyglet.event framework, so you can register and dispatch events in the same way you would with `pyglet.window.Window`, or any other `EventDispatcher` subclass.

Variables

- **game** – The game that this box will work with.
- **start** – The coordinates at the bottom rear left corner of this box.
- **end** – The coordinates at the top front right corner of this box.
- **name** – An optional name for this box.
- **surface_sound** – The sound that should be heard when walking in this box.
- **wall_sound** – The sound that should be heard when colliding with walls in this box.
- **type** – The type of this box.
- **data** – Arbitrary data for this box.

- **bounds** – The bounds of this box.
- **centre** – The point that lies at the centre of this box.
- **reverb** – The reverb that is assigned to this box.

close() → None

Close the attached door.

If this box is a door, set the `open` attribute of its `data` to `False`, and play the appropriate sound. Otherwise, raise `earwax.NotADoor`.

Parameters **door** – The door to close.

contains_point(*coordinates: earwax.point.Point*) → bool

Return whether or not this box contains the given point.

Returns `True` if this box spans the given coordinates, `False` otherwise.

Parameters **coordinates** – The coordinates to check.

could_fit(*box: earwax.mapping.box.Box*) → bool

Return whether or not the given box could be contained by this one.

Returns `True` if the given box could be contained by this box, `False` otherwise.

This method behaves like the `contains_point()` method, except that it works with `Box` instances, rather than `Point` instances.

This method doesn't care about the `parent` attribute on the given box. This method simply checks that the `start` and `end` points would fit inside this box.

Parameters **box** – The box whose bounds will be checked.

classmethod create_fitted(*game: Game, children: List[Box], pad_start: Optional[earwax.point.Point] = None, pad_end: Optional[earwax.point.Point] = None, **kwargs*) → Box

Return a box that fits all of `children` inside itself.

Pass a list of `Box` instances, and you'll get a box with its `start`, and `end` attributes set to match the outer bounds of the provided children.

You can use `pad_start`, and `pad_end` to add or subtract from the calculated start and end coordinates.

Parameters

- **children** – The list of `Box` instances to encapsulate.
- **pad_start** – A point to add to the calculated start coordinates.
- **pad_end** – A point to add to the calculated end coordinates.
- **kwargs** – The extra keyword arguments to pass to `Box.__init__`.

classmethod create_row(*game: Game, start: earwax.point.Point, size: earwax.point.Point, count: int, offset: earwax.point.Point, get_name: Optional[Callable[[int], str]] = None, on_create: Optional[Callable[[Box], None]] = None, **kwargs*) → List[Box]

Generate a list of boxes.

This method is useful for creating rows of buildings, or rooms on a corridor to name a couple of examples.

It can be used like so:

```
offices = Box.create_row(
    game,  # Every Box instance needs a game.
    Point(0, 0),  # The bottom_left corner of the first box.
    Point(3, 2, 0),  # The size of each box.
    3,  # The number of boxes to build.
    # The next argument is how far to move from the top right
    # corner of each created box:
    Point(1, 0, 0),
    # We want to name each room. For that, there is a function!
    get_name=lambda i: f'Room {i + 1}',
    # Let's make them all rooms.
    type=RoomTypes.room
)
```

This will result in a list containing 3 rooms:

- The first from (0, 0, 0) to (2, 1, 0)
- The second from (3, 0, 0) to (5, 1, 0)
- And the third from (6, 0, 0) to (8, 1, 0)

PLEASE NOTE: If none of the size coordinates are `>= 1`, the top right coordinate will be less than the bottom left, so `get_containing_box()` won't ever find it.

Parameters

- **`start`** – The `start` coordinate of the first box.
- **`size`** – The size of each box.
- **`count`** – The number of boxes to build.
- **`offset`** – The distance between the boxes.

  If no coordinate of the given value is `>= 1`, overlaps will occur.
- **`get_name`** – A function which should return an appropriate name.

  This function will be called with the current position in the loop.

  0 for the first room, 1 for the second, and so on.
- **`on_create`** – A function which will be called after each box is created.

  The only provided argument will be the box that was just created.
- **`kwargs`** – Extra keyword arguments to be passed to `Box.__init__`.

`get_nearest_point`(*point: earwax.point.Point*) → earwax.point.Point

Return the point on this box nearest to the provided point.

Parameters **`point`** – The point to start from.

`handle_door`() → None

Open or close the door attached to this box.

`handle_portal`() → None

Activate a portal attached to this box.

`is_door`

Return `True` if this box is a door.

`is_portal`

Return `True` if this box is a portal.

is_wall(*p: earwax.point.Point*) → bool
Return `True` if the provided point is inside a wall.

Parameters **p** – The point to interrogate.

on_activate() → None
Handle the enter key.

This event is dispatched when the player presses the enter key.

It is guaranteed that the instance this event is dispatched on is the one the player is stood on.

on_close() → None
Handle this box being closed.

on_collide(*coordinates: earwax.point.Point*) → None
Play an appropriate wall sound.

This function will be called by the Pyglet event framework, and should be called when a player collides with this box.

on_footstep(*bearing: float*, *coordinates: earwax.point.Point*) → None
Play an appropriate surface sound.

This function will be called by the Pyglet event framework, and should be called when a player is walking on this box.

This event is dispatched by `earwax.BoxLevel.move` upon a successful move.

Parameters **coordinates** – The coordinates the player has just moved to.

on_open() → None
Handle this box being opened.

open() → None
Open the attached door.

If this box is a door, set the `open` attribute of its `data` to `True`, and play the appropriate sound. Otherwise, raise `earwax.NotADoor`.

Parameters **box** – The box to open.

scheduled_close(*dt: float*) → None
Call `close()`.

This method will be called by `pyglet.clock.schedule_once`.

Parameters **dt** – The `dt` parameter expected by Pyglet's schedule functions.

sound_manager
Return a suitable sound manager.

class earwax.mapping.**BoxBounds**(*bottom_back_left: earwax.point.Point*, *top_front_right: earwax.point.Point*)
Bases: `object`

Bounds for a `earwax.Box` instance.

Variables

- **bottom_back_left** – The bottom back left point.
- **top_front_right** – The top front right point.
- **bottom_front_left** – The bottom front left point.
- **bottom_front_right** – The bottom front right point.

- **`bottom_back_right`** – The bottom back right point.
- **`top_back_left`** – The top back left point.
- **`top_front_left`** – The top front left point.
- **`top_back_right`** – The top back right point.

`area`
Return the area of the box.

`depth`
Get the depth of this box (front to back).

`height`
Return the height of this box.

`is_edge` (*p: earwax.point.Point*) → bool
Return `True` if `p` represents an edge.

Parameters **`p`** – The point to interrogate.

`volume`
Return the volume of this box.

`width`
Return the width of this box.

`class` `earwax.mapping.`**`BoxTypes`**
Bases: `enum.Enum`

The type of a box.

Variables

- **`empty`** – Empty space.

  Boxes of this type can be traversed wit no barriers.

- **`room`** – An open room with walls around the edge.

  Boxes of this type can be entered by means of a door. The programmer must provide some means of exit.

- **`solid`** – Signifies a solid, impassible barrier.

  Boxes of this type cannot be traversed.

`empty = 0`

`room = 1`

`solid = 2`

`exception` `earwax.mapping.`**`NotADoor`**
Bases: *`earwax.mapping.box.BoxError`*

The current box is not a door.

`exception` `earwax.mapping.`**`NotAPortal`**
Bases: *`earwax.mapping.box.BoxError`*

The current box is not a portal.

class earwax.mapping.**BoxLevel**(*game: Game*, *boxes: List[earwax.mapping.box.Box[typing.Any][Any]] = NOTHING*, *coordinates: earwax.point.Point = NOTHING*, *bearing: int = 0*, *current_box: Optional[earwax.mapping.box_level.CurrentBox] = None*)

Bases: `earwax.level.Level`

A level that deals with sound generation for boxes.

This level can be used in your games. Simply bind the various action methods (listed below) to whatever triggers suit your purposes.

Some of the attributes of this class refer to a "perspective". This could theoretically be anything you want, but most likely refers to the player. Possible exceptions include if you made an instance to represent some kind of long range vision for the player.

Action-ready Methods

- `move()`.
- `show_coordinates()`
- `show_facing()`
- `turn()`
- `show_nearest_door()`
- `describe_current_box()`

Variables

- **box** – The box that this level will work with.
- **coordinates** – The coordinates of the perspective.
- **bearing** – The direction the perspective is facing.
- **current_box** – The most recently walked over box.

  If you don't set this attribute when creating the instance, then the first time the player moves using the `move()` method, the name of the box they are standing on will be spoken.
- **reverb** – An optional reverb to play sounds through.

  You shouldn't write to this property, instead use the `connect_reverb()` method to set a new reverb, and `disconnect_reverb()` to clear.

activate(*door_distance: float = 2.0*) → Callable[[], None]

Return a function that can be call when the enter key is pressed.

First we check if the current box is a portal. If it is, then we call `handle_portal()`.

If it is not, we check to see if there is a door close enough to be opened or closed. If there is, then we call `handle_door()` on it.

If none of this works, and there is a current box, dispatch the `on_activate()` event to let the box do its own thing.

Parameters **door_distance** – How close doors have to be for this method to open or close them.

add_box(*box: earwax.mapping.box.Box[typing.Any][Any]*) → None

Add a box to `self.boxes`.

Parameters **box** – The box to add.

add_boxes(*boxes: Iterable[earwax.mapping.box.Box]*) → None
Add multiple boxes with one call.

Parameters **boxes** – An iterable for boxes to add.

add_default_actions() → None
Add some default actions.

This method adds the following actions:

- Move forward: W
- Turn 180 degrees: S
- Turn 45 degrees left: A
- Turn 45 degrees right: D
- Show coordinates: C
- Show the facing direction: F
- Describe current box: X
- Speak nearest door: Z
- Activate nearby objects: Return

calculate_coordinates(*distance: float*, *bearing: int*) → Tuple[float, float]
Calculate coordinates at the given distance in the given direction.

Used by `move()` to calculate new coordinates.

Override this method if you want to change the algorithm used to calculate the target coordinates.

Please bear in mind however, that the coordinates this method returns should always be 2d.

Parameters

- **distance** – The distance which should be used.
- **bearing** – The bearing the new coordinates are in.

  This value may not be the same as `self.bearing`.

collide(*box: earwax.mapping.box.Box[typing.Any][Any], coordinates: earwax.point.Point*) → None
Handle collitions.

Called to run collision code on a box.

Parameters

- **box** – The box the player collided with.
- **coordinates** – The coordinates the player was trying to reach.

describe_current_box() → None
Describe the current box.

get_angle_between(*other: earwax.point.Point*) → float
Return the angle between the perspective and the other coordinates.

This function takes into account `self.bearing`.

Parameters **other** – The target coordinates.

get_boxes(*t: Any*) → List[earwax.mapping.box.Box]
Return a list of boxes of the current type.

If no boxes are found, an empty list is returned.

Parameters **t** – The type of the boxes.

get_containing_box(*coordinates: earwax.point.Point*) → Optional[earwax.mapping.box.Box]
Return the box that spans the given coordinates.

If no box is found, `None` will be returned.

This method scans `self.boxes` using the `sort_boxes()` method.

Parameters **coordinates** – The coordinates the box should span.

get_current_box() → Optional[earwax.mapping.box.Box]
Get the box that lies at the current coordinates.

handle_box(*box: earwax.mapping.box.Box[typing.Any][Any]*) → None
Handle a bulk standard box.

The coordinates have already been set, and the `on_footstep` event dispatched, so all that is left is to speak the name of the new box, if it is different to the last one, update `self.reverb` if necessary, and store the new box.

move(*distance: float = 1.0*, *vertical: Optional[float] = None*, *bearing: Optional[int] = None*) → Callable[[], None]
Return a callable that allows the player to move on the map.

If the move is successful (I.E.: There is a box at the destination coordinates), the `on_move()` event is dispatched.

If not, then `on_move_fail()` is dispatched.

Parameters

- **distance** – The distance to move.
- **vertical** – An optional adjustment to be added to the vertical position.
- **bearing** – An optional direction to move in.

  If this value is `None`, then `self.bearing` will be used.

nearest_by_type(*start: earwax.point.Point*, *data_type: Any*, *same_z: bool = True*) → Optional[earwax.mapping.box_level.NearestBox]
Get the nearest box to the given point by type.

If no boxes of the given type are found, `None` will be returned.

Parameters

- **start** – The point to start looking from.
- **data_type** – The type of `box data` to search for.
- **same_z** – If this value is `True`, only boxes on the same z axis will be considered.

nearest_door(*start: earwax.point.Point*, *same_z: bool = True*) → Optional[earwax.mapping.box_level.NearestBox]
Get the nearest door.

Iterates over all doors, and returned the nearest one.

Parameters

- **start** – The coordinates to start from.

- **`same_z`** – If `True`, then doors on different levels will not be considered.

`nearest_portal`(*start: earwax.point.Point*, *same_z: bool = True*) → Optional[earwax.mapping.box_level.NearestBox]
Return the nearest portal.

Parameters

- **`start`** – The coordinates to start from.
- **`same_z`** – If `True`, then portals on different levels will not be considered.

`on_move_fail`(*distance: float, vertical: Optional[float], bearing: int, coordinates: earwax.point.Point*) → None
Handle a move failure.

An event that will be dispatched when the `move()` action has been used, but no move was performed.

Parameters

- **`distance`** – The `distance` value that was passed to `move()`.
- **`vertical`** – The `vertical` value that was passed to `move`.
- **`bearing`** – The `bearing` argument that was passed to `move`, or `self.bearing`.

`on_move_success`() → None
Handle a successful move.

An event that will be dispatched when the `move()` action is used.

By default, this method plays the correct footstep sound.

`on_push`() → None
Set listener orientation, and start ambiances and tracks.

`on_turn`() → None
Handle turning.

An event that will dispatched when the `turn()` action is used.

`register_box`(*box: earwax.mapping.box.Box*) → None
Register a box that is already in the boxes list.

Parameters **`box`** – The box to register.

`remove_box`(*box: earwax.mapping.box.Box[typing.Any][Any]*) → None
Remove a box from `self.boxes`.

Parameters **`box`** – The box to remove.

`set_bearing`(*angle: int*) → None
Set the direction of travel and the listener's orientation.

Parameters **`angle`** – The bearing (in degrees).

`set_coordinates`(*p: earwax.point.Point*) → None
Set the current coordinates.

Also set listener position.

Parameters **`p`** – The new point to assign to `self.coordinates`.

`show_coordinates`(*include_z: bool = False*) → Callable[[], None]
Speak the current coordinates.

show_facing(*include_angle: bool = True*) → Callable[[], None]

Return a function that will let you see the current bearing as text.

For example:

```
l = BoxLevel(...)
l.action('Show facing', symbol=key.F)(l.show_facing())
```

Parameters **include_angle** – If `True`, then the actual angle will be shown along with the direction name.

show_nearest_door(*max_distance: Optional[float] = None*) → Callable[[], None]

Return a callable that will speak the position of the nearest door.

Parameters **max_distance** – The maximum distance between the current coordinates and the nearest door where the door will still be reported.

If this value is `None`, then any door will be reported.

sort_boxes() → List[earwax.mapping.box.Box]

Return `children` sorted by area.

turn(*amount: int*) → Callable[[], None]

Return a turn function.

Return a function that will turn the perspective by the given amount and dispatch the `on_turn` event.

For example:

```
l = BoxLevel(...)
l.action('Turn right', symbol=key.D)(l.turn(45))
l.action('Turn left', symbol=key.A)(l.turn(-45))
```

The resulting angle will always be in the range 0-359.

Parameters **amount** – The amount to turn by.

Positive numbers turn clockwise, while negative numbers turn anticlockwise.

class earwax.mapping.**CurrentBox**(*coordinates: earwax.point.Point, box: earwax.mapping.box.Box[typing.Any][Any]*)

Bases: `object`

Store a reference to the current box.

This class stores the position too, so that caching can be performed.

Variables

- **coordinates** – The coordinates that were last checked.
- **box** – The last current box.

class earwax.mapping.**NearestBox**(*box: earwax.mapping.box.Box, coordinates: earwax.point.Point, distance: float*)

Bases: `object`

A reference to the nearest box.

Variables

- **box** – The box that was found.
- **coordinates** – The nearest coordinates to the ones specified.

- **distance** – The distance between the supplied coordinates, and `coordinates`.

class earwax.mapping.**Door**(*open: bool = True, closed_sound: Optional[pathlib.Path] = None, open_sound: Optional[pathlib.Path] = None, close_sound: Optional[pathlib.Path] = None, close_after: Union[float, Tuple[float, float], None] = None, can_open: Optional[Callable[[], bool]] = None, can_close: Optional[Callable[[], bool]] = None*)

Bases: `object`

An object that can be added to a box to optionally block travel.

Doors can currently either be open or closed. When opened, they can optionally close after a specified time:

```
Door()  # Standard open door.
Door(open=False)  # Closed door.
Door(close_after=5.0)  # Will automatically close after 5 seconds.
# A door that will automatically close between 5 and 10 seconds after
# it has been opened:
Door(close_after=(5.0, 10.0)
```

Variables

- **open** – Whether or not this box can be walked on.

  If this value is `False`, then the player will hear `closed_sound` when trying to walk on this box.

  If this value is `True`, the player will be able to enter the box as normal.
- **closed_sound** – The sound that will be heard if `open` is `False`.
- **open_sound** – The sound that will be heard when opening this door.
- **close_sound** – The sound that will be heard when closing this door.
- **close_after** – When (if ever) to close the door after it has been opened.

  This attribute supports 3 possible values:

  – `None`: The door will not close on its own.

  – **A tuple of two positive floats a and b: A random number** between `a` and `b` will be selected, and the door will automatically close after that time.

  – A float: The exact time the door will automatically close after.
- **can_open** – An optional method which will be used to decide whether or not this door can be opened at this time.

  This method must return `True` or `False`, and must handle any messages which should be sent to the player.
- **can_close** – An optional method which will be used to decide whether or not this door can be closed at this time.

  This method must return `True` or `False`, and must handle any messages which should be sent to the player.

class earwax.mapping.**MapEditor**(*game: Game, boxes: List[earwax.mapping.box.Box[typing.Any][Any]] = NOTHING, coordinates: earwax.point.Point = NOTHING, bearing: int = 0, current_box: Optional[earwax.mapping.box_level.CurrentBox] = None, filename: Optional[pathlib.Path] = None, context: earwax.mapping.map_editor.MapEditorContext = NOTHING*)

Bases: `earwax.mapping.box_level.BoxLevel`

A level which can be used for editing maps.

When this level talks about a map, it talks about a `earwax.mapping.map_editor.LevelMap` instance.

`box_menu`(*box: earwax.mapping.map_editor.MapEditorBox*) → None
: Push a menu to configure the provided box.

`box_sound`(*template: earwax.mapping.map_editor.BoxTemplate*, *name: str*) → Callable[[], Generator[None, None, None]]
: Push an editor for setting the given sound.

Parameters

- **`template`** – The template to modify.
- **`name`** – The name of the sound to modify.

`box_sounds`() → None
: Push a menu for configuring sounds.

`boxes_menu`() → None
: Push a menu to select a box to configure.

If there is only 1 box, it will not be shown.

`complain_box`() → None
: Complain about there being no box.

`create_box`() → None
: Create a box, then call `box_menu()`.

`get_default_context`() → earwax.mapping.map_editor.MapEditorContext
: Return a suitable context.

`id_box`() → Generator[None, None, None]
: Change the ID for the current box.

`label_box`() → Generator[None, None, None]
: Rename the current box.

`on_move_fail`(*distance: float, vertical: Optional[float], bearing: int, coordinates: earwax.point.Point*) → None
: Tell the user their move failed.

`point_menu`(*template: earwax.mapping.map_editor.BoxTemplate*, *point: earwax.mapping.map_editor.BoxPoint*) → Callable[[], None]
: Push a menu for configuring individual points.

`points_menu`() → None
: Push a menu for moving the current box.

`rename_box`() → Generator[None, None, None]
: Rename the current box.

`save`() → None
: Save the map level.

class `earwax.mapping.`**`MapEditorContext`**(*level: MapEditor*, *level_map: earwax.mapping.map_editor.LevelMap*, *template_ids: Dict[str, earwax.mapping.map_editor.BoxTemplate] = NOTHING*, *box_ids: Dict[str, earwax.mapping.box.Box[str][str]] = NOTHING*)

Bases: `object`

A context to hold map information.

This class acts as an interface between a `LevelMap` instance, and a `MapEditor` instance.

add_template(*template: earwax.mapping.map_editor.BoxTemplate*, *box: Optional[earwax.mapping.map_editor.MapEditorBox] = None*) → None

Add a template to this context.

This method will add the given template to its `box_template_ids` dictionary.

Parameters **template** – The template to add.

reload_template(*template: earwax.mapping.map_editor.BoxTemplate*) → None

Reload the given template.

This method recreates the box associated with the given template.

Parameters **template** – The template to reload.

to_box(*template: earwax.mapping.map_editor.BoxTemplate*) → earwax.mapping.map_editor.MapEditorBox

Return a box from a template.

Parameters **template** – The template to convert.

to_point(*data: earwax.mapping.map_editor.BoxPoint*) → earwax.point.Point

Return a point from the given data.

Parameters **data** – The `BoxPoint` to load the point from.

class earwax.mapping.**Portal**(*level: BoxLevel*, *coordinates: earwax.point.Point*, *bearing: Optional[int] = None*, *enter_sound: Optional[pathlib.Path] = None*, *exit_sound: Optional[pathlib.Path] = None*, *can_use: Optional[Callable[[], bool]] = None*)

Bases: `earwax.mixins.RegisterEventMixin`

A portal to another map.

An object that can be added to a `earwax.Box` to make a link between two maps.

This class implements `pyglet.event.EventDispatcher`, so events can be registered and dispatched on it.

The currently-registered events are:

- `on_enter()`
- `on_exit()`

Variables

- **level** – The destination level.
- **coordinates** – The exit coordinates.
- **bearing** – If this value is `None`, then it will be used for the player's bearing after this portal is used. Otherwise, the bearing from the old level will be used.
- **enter_sound** – The sound that should play when entering this portal.

  This sound is probably only used when an NPC uses the portal.
- **exit_sound** – The sound that should play when exiting this portal.

  This is the sound that the player will hear when using the portal.

- **can_use** – An optional method which will be called to ensure that this portal can be used at this time.

  This function should return `True` or `False`, and should handle any messages which should be sent to the player.

on_enter() → None
Handle a player entering this portal.

on_exit() → None
Handle a player exiting this portal.

earwax.menus package

Submodules

earwax.menus.action_menu module

Provides the ActionMenu class.

class earwax.menus.action_menu.**ActionMenu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, input_mode: Optional[earwax.input_modes.InputModes] = NOTHING, all_triggers_label: Optional[str] = '<< Show all triggers >>'*)

Bases: *earwax.menus.menu.Menu*

A menu to show a list of actions and their associated triggers.

You can use this class with any game, like so:

```
from earwax import Game, Level, ActionMenu
from pyglet.window import Window, key
w = Window(caption='Test Game')
g = Game()
l = Level()
@l.action('Show actions', symbol=key.SLASH, modifiers=key.MOD_SHIFT)
def actions_menu():
    '''Show an actions menu.'''
    a = ActionMenu(g, 'Actions')
    g.push_level(a)

g.push_level(l)
g.run(w)
```

Now, if you press shift and slash (a question mark on english keyboards), you will get an action menu.

This code can be shortened to:

```
@l.action('Show actions', symbol=key.SLASH, modifiers=key.MOD_SHIFT)
def actions_menu():
    '''Show an actions menu.'''
    game.push_action_menu()
```

If you want to override how triggers appear in the menu, then you can override *symbol_to_string()* and *mouse_to_string()*.

Variables

- **input_mode** – The input mode this menu will show actions for.
- **all_triggers_label** – The label for the "All triggers" entry.

  If this value is `None` no such entry will be shown.

action_menu(*action: earwax.action.Action*) → Callable[[], Optional[Generator[None, None, None]]]
Show a submenu of triggers.

Override this method to change how the submenu for actions is displayed.

Parameters action – The action to generate the menu for.

action_title(*action: earwax.action.Action, triggers: List[str]*) → str
Return a suitable title for the given action.

This method is used when building the menu when `input_mode` is not `None`.

Parameters

- **action** – The action whose name will be used.
- **triggers** – A list of triggers gleaned from the given action.

get_default_input_mode() → earwax.input_modes.InputModes
Get the default input mode.

handle_action(*action: earwax.action.Action*) → Callable[[], Optional[Generator[None, None, None]]]
Handle an action.

This method is used as the menu handler that is triggered when you select a trigger to activate the current action.

Parameters action – The action to run.

hat_direction_to_string(*direction: Tuple[int, int]*) → str
Return the given hat direction as a string.

mouse_to_string(*action: earwax.action.Action*) → str
Describe how to trigger the given action with the mouse.

Returns a string representing the mouse button and modifiers needed to trigger the provided action.

You must be certain that `action.mouse_button is not None`.

Override this method to change how mouse triggers appear.

Parameters action – The action whose `mouse_button` attribute this method will be working on.

show_all() → None
Show all triggers.

symbol_to_string(*action: earwax.action.Action*) → str
Describe how to trigger the given action with the keyboard.

Returns a string representing the symbol and modifiers needed to trigger the provided action.

You must be certain that `action.symbol is not None`.

Override this method to change how symbol triggers appear.

Parameters **action** – The action whose `symbol` attribute this method will be working on.

earwax.menus.config_menu module

Provides the ConfigMenu class,.

class earwax.menus.config_menu.**ConfigMenu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, config: earwax.config.Config = NOTHING*)
Bases: `earwax.menus.menu.Menu`

A menu that allows the user to set values on configuration sections.

If an option is present with a type the menu doesn't know how to handle, `earwax.UnknownTypeError` will be raised.

Variables

- **config** – The configuration section this menu will configure.
- **type_handlers** – Functions to handle the types this menu knows about.

  New types can be handled with the `type_handler()` method.

activate_handler(*handler: earwax.menus.config_menu.TypeHandler, option: earwax.config.ConfigValue*) → Callable[[], Optional[Generator[None, None, None]]]
Activates the given handler with the given configuration value.

Used by the `option_menu()` method when building menus.

Parameters

- **handler** – The `TypeHandler` instance that should be activated.
- **option** – The `ConfigValue` instance the handler should work with.

clear_value(*option: earwax.config.ConfigValue*) → None
Clear the value.

Sets `option.value` to `None`.

Used by the default `TypeHandler` that handles nullable values.

Parameters **option** – The `ConfigValue` instance whose value should be set to `None`.

earwax_config() → earwax.config.Config
Return the main earwax configuration.

get_option_name(*option: earwax.config.ConfigValue*, *name: str*) → str
Get the name for the given option.

The provided `name` argument will be the attribute name, so should only be used if the option has no `__section_name__` attribute.

Parameters

- **option** – The `ConfigValue` instance whose name should be returned.
- **name** – The name of the attribute that holds the option.

get_subsection_name(*subsection: earwax.config.Config*, *name: str*) → str
Get the name for the given subsection.

The provided `name` argument will be the attribute name, so should only be used if the subsection has no `__section_name__` attribute.

Parameters

- **subsection** – The `Config` instance whose name should be returned.
- **name** – The name of the attribute that holds the subsection.

handle_bool(*option: earwax.config.ConfigValue*) → None
Toggle a boolean value.

Used by the default `TypeHandler` that handles boolean values.

Parameters **option** – The `ConfigValue` instance to work on.

handle_float(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow editing floats.

Used by the default `TypeHandler` that handles float values.

Parameters **option** – The `ConfigValue` instance to work on.

handle_int(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow editing integers.

Used by the default `TypeHandler` that handles integer values.

Parameters **option** – The `ConfigValue` instance to work on.

handle_path(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow selecting files and folders.

Used by the default `TypeHandler` that handles `pathlib.Path` values.

Parameters **option** – The `ConfigValue` instance to work on.

handle_string(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow editing strings.

Used by the default `TypeHandler` that handles string values.

Parameters **option** – The `ConfigValue` instance to work on.

option_menu(*option: earwax.config.ConfigValue*, *name: str*) → Callable[[], Generator[None, None, None]]
Add a menu for the given option.

If the type of the provided option is a `Union` type (like `Optional[str]`), then an entry for editing each type will be added to the menu. Otherwise, there will be only one entry.

The only special case is when the type is a tuple of values. If this happens, the menu will instead be populated with a list of entries corrisponding to the values of the tuple.

At the end of the menu, there will be an option to restore the default value.

Parameters

- **`option`** – The `ConfigValue` instance to generate a menu for.
- **`name`** – The proper name of the given option, as returned by `get_option_name()`.

`set_value`(*option: earwax.config.ConfigValue*, *value: Any*, *message: str = 'Done.'*) → Callable[[], None]

Set a value.

Returns a callable that can be used to set the value of the provided option to the provided value.

This method returns a callable because it is used extensively by `option_menu()`, and a bunch of lambdas becomes less readable. Plus, Mypy complains about them.

Parameters

- **`option`** – The `ConfigValue` instance to work on.
- **`value`** – The value to set `option.value` to.
- **`message`** – The message to be spoken after setting the value.

`subsection_menu`(*subsection: earwax.config.Config*, *name: str*) → Callable[[], Generator[None, None, None]]

Add a menu for the given subsection.

By default, creates a new `earwax.ConfigMenu` instance, and returns a function that - when called - will push it onto the stack.

Parameters

- **`subsection`** – The `Config` instance to create a menu for.
- **`name`** – The proper name of the subsection, returned by `get_subsection_name()`.

`type_handler`(*type_: object*, *title: Callable[[earwax.config.ConfigValue, str], str]*) → Callable[[Callable[[earwax.config.ConfigValue], Optional[Generator[None, None, None]]]], Callable[[earwax.config.ConfigValue], Optional[Generator[None, None, None]]]]

Add a type handler.

Decorate a function to be used as a type handler:

```python
from datetime import datetime, timedelta
from earwax import ConfigMenu, tts

m = ConfigMenu(pretend_config, 'Options', game)

@m.type_handler(datetime, lambda option, name: 'Add a week')
def add_week(option):
    '''Add a week to the current value.'''
    option.value += timedelta(days=7)
    self.game.output('Added a week.')
    m.game.pop_level()
```

Handlers can do anything menu item functions can do, including creating more menus, and yielding.

Parameters

- **`type`** – The type this handler should be registered for.
- **`title`** – A function which will return the title for the menu item for this handler.

class earwax.menus.config_menu.**TypeHandler**(*title: Callable[[earwax.config.ConfigValue, str], str], func: Callable[[earwax.config.ConfigValue], Optional[Generator[None, None, None]]]*)

Bases: `object`

A type handler for use with `ConfigMenu` instances.

Variables

- **title** – A function that will return a string which can be used as the title for the menu item generated by this handler.
- **func** – The function that will be called when this handler is required.

exception earwax.menus.config_menu.**UnknownTypeError**

Bases: `Exception`

An unknown type was encountered.

An exception which will be thrown if a `ConfigMenu` instance doesn't know how to handle the given type.

earwax.menus.file_menu module

Provides the FileMenu class.

class earwax.menus.file_menu.**FileMenu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, path: pathlib.Path = NOTHING, func: Callable[[Optional[pathlib.Path]], Optional[Generator[None, None, None]]] = <built-in function print>, root: Optional[pathlib.Path] = None, empty_label: Optional[str] = None, directory_label: Optional[str] = None, show_directories: bool = True, show_files: bool = True, up_label: str = '..'*)

Bases: *earwax.menus.menu.Menu*

A menu for selecting a file.

File menus can be used as follows:

```python
from pathlib import Path
from earwax import Game, Level, FileMenu, tts
from pyglet.window import key, Window
w = Window(caption='Test Game')
g = Game()
l = Level(g)
@l.action('Show file menu', symbol=key.F)
def file_menu():
    '''Show a file menu.'''
    def inner(p):
        tts.speak(str(p))
        g.pop_level()
    f = FileMenu(g, 'File Menu', Path.cwd(), inner)
    g.push_level(f)
```

(continues on next page)

(continued from previous page)

```
g.push_level(l)
g.run(w)
```

Variables

- **`path`** – The path this menu will start at.
- **`func`** – The function to run with the resulting file or directory.
- **`root`** – The root directory which this menu will be chrooted to.
- **`empty_label`** – The label given to an entry which will allow this menu to return None as a result.

  If this label is None (the default), then then no such option will be available.
- **`directory_label`** – The label given to an entry which will allow a directory - in addition to files - to be selected.

  If this argument is None (the default), then no such option will be available.

  If you only want directories to be selected, then pass show_files=False to the constructor.
- **`show_directories`** – Whether or not to show directories in the list.
- **`show_files`** – Whether or not to include files in the list.
- **`up_label`** – The label given to the entry to go up in the directory tree.

`navigate_to`(*path: pathlib.Path*) → Callable[[], None]

Navigate to a different path.

Instead of completely replacing the menu, just change the path, and re- use this instance.

`rebuild_menu`() → None

Rebuild the menu.

This method will be called once after initialisation, and every time the directory is changed by the `navigate_to()` method.

`select_item`(*path: Optional[pathlib.Path]*) → Callable[[], Optional[Generator[None, None, None]]]

Select an item.

Used as the menu handler in place of a lambda.

Parameters **`path`** – The path that has been selected. Could be a file or a directory.

earwax.menus.menu module

Provides the Menu class.

class earwax.menus.menu.**Menu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0*)

Bases: *`earwax.level.Level`*, *`earwax.mixins.TitleMixin`*, *`earwax.mixins.DismissibleMixin`*

A menu of `MenuItem` instances.

Menus hold multiple menu items which can be activated using actions.

As menus are simply *Level* subclasses, they can be *pushed*, *popped*, and *replaced*.

To add items to a menu, you can either use the *item()* decorator, or the *add_item()* function.

Here is an example of both methods:

```
from earwax import Game, Level, Menu
from pyglet.window import key, Window
w = Window(caption='Test Game')
g = Game()
l = Level()
@l.action('Show menu', symbol=key.M)
def menu():
    '''Show a menu with 2 items.'''
    m = Menu(g, 'Menu')
    @m.item(title='First Item')
    def first_item():
        g.output('First menu item.')
        g.pop_level()
    def second_item():
        g.output('Second menu item.')
        g.pop_level()
    m.add_item(second_item, title='Second Item')
    g.push_level(m)

g.push_level(l)
g.run(w)
```

To override the default actions that are added to a menu, subclass `earwax.Menu`, and override `__attrs_post_init__()`.

Variables

- **`item_sound_path`** – The default sound to play when moving through the menu.

  If the selected item's `sound_path` attribute is not `None`, then that value takes precedence.
- **`items`** – The list of MenuItem instances for this menu.
- **`position`** – The user's position in this menu.
- **`search_timeout`** – The maximum time between menu searches.
- **`search_time`** – The time the last menu search was performed.
- **`search_string`** – The current menu search search string.

`activate``()` → Optional[Generator[None, None, None]]

Activate the currently focused menu item.

Usually triggered by the enter key.

`add_item``(`*func: Callable[[], Optional[Generator[None, None, None]]], **kwargs*`)` → earwax.menus.menu_item.MenuItem

Add an item to this menu.

For example:

```
m = Menu(game, 'Example Menu')
def f():
    game.output('Menu item activated.')
```

(continues on next page)

(continued from previous page)

```
m.add_item(f, title='Test Item')
m.add_item(f, sound_path=Path('sound.wav'))
```

If you would rather use decorators, use the `item()` method instead.

Parameters

- **func** – The function which will be called when the menu item is selected.
- **kwargs** – Extra arguments to be passed to the constructor of `earwax.MenuItem`.

add_submenu(*menu: earwax.menus.menu.Menu*, *replace: bool*, ***kwargs*) → earwax.menus.menu_item.MenuItem

Add a submenu to this menu.

Parameters

- **menu** – The menu to show when the resulting item is activated.
- **replace** – If `True`, then the new menu will replace this one in the levels stack.
- **kwargs** – The additional arguments to pass to `add_item()`.

current_item

Return the currently selected menu item.

If position is -1, return `None`.

end() → None

Move to the end of a menu.

Usually triggered by the end key.

classmethod from_credits(*game: Game, credits: List[earwax.credit.Credit], title: str = 'Game Credits'*) → Menu

Return a menu for showing credits.

Parameters

- **game** – The game to use.
- **credits** – The credits to show.
- **title** – The title of the new menu.

home() → None

Move to the start of a menu.

Usually triggered by the home key.

item(***kwargs*) → Callable[[Callable[[], Optional[Generator[None, None, None]]]], earwax.menus.menu_item.MenuItem]

Decorate a function to be used as a menu item.

For example:

```
@menu.item(title='Title')
def func():
    pass

@menu.item(sound_path=Path('sound.wav'))
def item_with_sound():
    pass
```

If you don't want to use a decorator, you can use the `add_item()` method instead.

Parameters **`kwargs`** – Extra arguments to be passed to the constructor of `earwax.MenuItem`.

`make_sound`(*item: earwax.menus.menu_item.MenuItem*, *path: pathlib.Path*) → earwax.sound.Sound
Return a sound object.

Parameters

- **`item`** – The menu item to make the sound for.

  This value is probably `current_item`.

- **`path`** – The path to load the sound from.

  This value will have been determined by `show_selection()`, and may have been loaded from the menu item itself, or the main earwax configuration.

`move_down`() → None
Move down in this menu.

Usually triggered by the down arrow key.

`move_up`() → None
Move up in this menu.

Usually triggered by the up arrow key.

`on_pop`() → None
Destroy `select_sound` if necessary.

`on_push`() → None
Handle this menu being pushed.

This method is called when this object has been pushed onto a `Game` instance.

By default, show the current selection. That will be the same as speaking the title, unless `self.position` has been set to something other than -1..

`on_reveal`() → None
Show selection again.

`on_text`(*text: str*) → None
Handle sent text.

By default, performs a search of this menu.

Parameters **`text`** – The text that has been sent.

`show_selection`() → None
Speak the menu item at the current position.

If `self.position` is -1, this method speaks `self.title`.

This function performs no error checking, so it will happily throw errors if `position` is something stupid.

`classmethod yes_no`(*game: Game, yes_action: Callable[[], Optional[Generator[None, None, None]]], no_action: Callable[[], Optional[Generator[None, None, None]]], title: str = 'Are you sure?', yes_label: str = 'Yes', no_label: str = 'No', **kwargs*) → Menu
Create and return a yes no menu.

Parameters

- **`game`** – The game to bind the new menu to.
- **`yes_action`** – The function to be called if the yes item is selected.

- **no_action** – The action to be performed if no is selected.
- **title** – The title of the menu.
- **yes_label** – The label of the yes item.
- **no_label** – The title of the no label.
- **kwargs** – Extra keyword arguments to be passed to the `Menu` constructor.

earwax.menus.menu_item module

Provides the MenuItem class.

class `earwax.menus.menu_item.`**MenuItem**(*func: Callable[[], Optional[Generator[None, None, None]]], title: Union[str, TitleFunction, None] = None, select_sound_path: Optional[pathlib.Path] = None, loop_select_sound: bool = False, activate_sound_path: Optional[pathlib.Path] = None*)

Bases: `earwax.mixins.RegisterEventMixin`

An item in a `Menu`.

This class is rarely used directly, instead `earwax.menu.Menu.add_item()` or `earwax.menu.Menu.item()` can be used to return an instance.

Variables

- **func** – The function which will be called when this item is activated.
- **title** – The title of this menu item.

  If this value is a callable, it should return a string which will be used as the title.
- **select_sound_path** – The path to a sound which should play when this menu item is selected.

  If this value is `None` (the default), then no sound will be heard unless the containing menu has its `item_select_sound_path` attribute set to something that is not `None`, or `earwax.EarwaxConfig.menus.default_item_select_sound` is not `None`.
- **activate_sound_path** – The path to a sound which should play when this menu item is activated.

  If this value is `None` (the default), then no sound will be heard unless the containing menu has its `item_activate_sound_path` attribute set to something that is not `None`, or `earwax.EarwaxConfig.menus.default_item_select_sound` is not `None`.

get_title() → Optional[str]

Return the proper title of this object.

If `self.title` is a callable, its return value will be returned.

on_selected() → None

Handle this menu item being selected.

earwax.menus.reverb_editor module

Provides the ReverbEditor class.

class earwax.menus.reverb_editor.**ReverbEditor**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, reverb: object = NOTHING, settings: earwax.reverb.Reverb = NOTHING, setting_items: List[earwax.menus.menu_item.MenuItem] = NOTHING*)

Bases: `earwax.menus.menu.Menu`

A menu for editing reverbs.

adjust_value(*amount: earwax.menus.reverb_editor.ValueAdjustments*) → Callable[[], None]
Restore the current menu item to the default.

edit_value(*setting: earwax.menus.reverb_editor.ReverbSetting*, *value: float*) → Callable[[], Generator[None, None, None]]
Edit the given value.

get_default_reverb() → object
Raise an error.

get_default_settings() → earwax.reverb.Reverb
Raise an error.

reset() → None
Reload this menu.

set_value(*setting: earwax.menus.reverb_editor.ReverbSetting*, *value: float*) → None
Set the value.

class earwax.menus.reverb_editor.**ReverbSetting**(*name: str*, *description: str*, *min: float*, *max: float*, *default: float*, *increment: float = 0.05*)

Bases: `object`

A setting for reverb.

class earwax.menus.reverb_editor.**ValueAdjustments**
Bases: `enum.Enum`

Possible value adjustments for menu actions.

decrement = 1

default = 0

increment = 2

Module contents

Provides all menu-related classes.

By default:

- **Menus are lists of items which can be traversed with the arrow keys, or by** searching.

- The first item can be focussed with the home key.
- The last item can be focussed with the end key.
- The selected item can be activated with the enter key.

Optionally, menus can be dismissed with the escape key.

class earwax.menus.**Menu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0*)

Bases: *earwax.level.Level*, *earwax.mixins.TitleMixin*, *earwax.mixins.DismissibleMixin*

A menu of `MenuItem` instances.

Menus hold multiple menu items which can be activated using actions.

As menus are simply *Level* subclasses, they can be *pushed*, *popped*, and *replaced*.

To add items to a menu, you can either use the *item()* decorator, or the *add_item()* function.

Here is an example of both methods:

```
from earwax import Game, Level, Menu
from pyglet.window import key, Window
w = Window(caption='Test Game')
g = Game()
l = Level()
@l.action('Show menu', symbol=key.M)
def menu():
    '''Show a menu with 2 items.'''
    m = Menu(g, 'Menu')
    @m.item(title='First Item')
    def first_item():
        g.output('First menu item.')
        g.pop_level()
    def second_item():
        g.output('Second menu item.')
        g.pop_level()
    m.add_item(second_item, title='Second Item')
    g.push_level(m)

g.push_level(l)
g.run(w)
```

To override the default actions that are added to a menu, subclass `earwax.Menu`, and override `__attrs_post_init__()`.

Variables

- **item_sound_path** – The default sound to play when moving through the menu.

  If the selected item's `sound_path` attribute is not `None`, then that value takes precedence.
- **items** – The list of MenuItem instances for this menu.
- **position** – The user's position in this menu.
- **search_timeout** – The maximum time between menu searches.
- **search_time** – The time the last menu search was performed.
- **search_string** – The current menu search search string.

activate() → Optional[Generator[None, None, None]]

Activate the currently focused menu item.

Usually triggered by the enter key.

add_item(*func: Callable[[], Optional[Generator[None, None, None]]], **kwargs*) → earwax.menus.menu_item.MenuItem

Add an item to this menu.

For example:

```
m = Menu(game, 'Example Menu')
def f():
    game.output('Menu item activated.')
m.add_item(f, title='Test Item')
m.add_item(f, sound_path=Path('sound.wav'))
```

If you would rather use decorators, use the `item()` method instead.

Parameters

- **func** – The function which will be called when the menu item is selected.
- **kwargs** – Extra arguments to be passed to the constructor of `earwax.MenuItem`.

add_submenu(*menu: earwax.menus.menu.Menu, replace: bool, **kwargs*) → earwax.menus.menu_item.MenuItem

Add a submenu to this menu.

Parameters

- **menu** – The menu to show when the resulting item is activated.
- **replace** – If `True`, then the new menu will replace this one in the levels stack.
- **kwargs** – The additional arguments to pass to `add_item()`.

current_item

Return the currently selected menu item.

If position is -1, return `None`.

end() → None

Move to the end of a menu.

Usually triggered by the end key.

classmethod from_credits(*game: Game, credits: List[earwax.credit.Credit], title: str = 'Game Credits'*) → Menu

Return a menu for showing credits.

Parameters

- **game** – The game to use.
- **credits** – The credits to show.
- **title** – The title of the new menu.

home() → None

Move to the start of a menu.

Usually triggered by the home key.

item(***kwargs*) → Callable[[Callable[[], Optional[Generator[None, None, None]]]], earwax.menus.menu_item.MenuItem]

Decorate a function to be used as a menu item.

For example:

```
@menu.item(title='Title')
def func():
    pass

@menu.item(sound_path=Path('sound.wav'))
def item_with_sound():
    pass
```

If you don't want to use a decorator, you can use the `add_item()` method instead.

Parameters **`kwargs`** – Extra arguments to be passed to the constructor of `earwax.MenuItem`.

`make_sound`(*item: earwax.menus.menu_item.MenuItem*, *path: pathlib.Path*) → earwax.sound.Sound
Return a sound object.

Parameters

- **`item`** – The menu item to make the sound for.

  This value is probably `current_item`.

- **`path`** – The path to load the sound from.

  This value will have been determined by `show_selection()`, and may have been loaded from the menu item itself, or the main earwax configuration.

`move_down`() → None
Move down in this menu.

Usually triggered by the down arrow key.

`move_up`() → None
Move up in this menu.

Usually triggered by the up arrow key.

`on_pop`() → None
Destroy `select_sound` if necessary.

`on_push`() → None
Handle this menu being pushed.

This method is called when this object has been pushed onto a *`Game`* instance.

By default, show the current selection. That will be the same as speaking the title, unless `self.position` has been set to something other than -1..

`on_reveal`() → None
Show selection again.

`on_text`(*text: str*) → None
Handle sent text.

By default, performs a search of this menu.

Parameters **`text`** – The text that has been sent.

`show_selection`() → None
Speak the menu item at the current position.

If `self.position` is -1, this method speaks `self.title`.

This function performs no error checking, so it will happily throw errors if `position` is something stupid.

classmethod yes_no(*game: Game, yes_action: Callable[[], Optional[Generator[None, None, None]]], no_action: Callable[[], Optional[Generator[None, None, None]]], title: str = 'Are you sure?', yes_label: str = 'Yes', no_label: str = 'No', **kwargs*) → Menu

Create and return a yes no menu.

Parameters

- **game** – The game to bind the new menu to.
- **yes_action** – The function to be called if the yes item is selected.
- **no_action** – The action to be performed if no is selected.
- **title** – The title of the menu.
- **yes_label** – The label of the yes item.
- **no_label** – The title of the no label.
- **kwargs** – Extra keyword arguments to be passed to the `Menu` constructor.

class earwax.menus.**MenuItem**(*func: Callable[[], Optional[Generator[None, None, None]]], title: Union[str, TitleFunction, None] = None, select_sound_path: Optional[pathlib.Path] = None, loop_select_sound: bool = False, activate_sound_path: Optional[pathlib.Path] = None*)

Bases: *earwax.mixins.RegisterEventMixin*

An item in a `Menu`.

This class is rarely used directly, instead `earwax.menu.Menu.add_item()` or `earwax.menu.Menu.item()` can be used to return an instance.

Variables

- **func** – The function which will be called when this item is activated.
- **title** – The title of this menu item.

  If this value is a callable, it should return a string which will be used as the title.
- **select_sound_path** – The path to a sound which should play when this menu item is selected.

  If this value is `None` (the default), then no sound will be heard unless the containing menu has its `item_select_sound_path` attribute set to something that is not `None`, or `earwax.EarwaxConfig.menus.default_item_select_sound` is not `None`.
- **activate_sound_path** – The path to a sound which should play when this menu item is activated.

  If this value is `None` (the default), then no sound will be heard unless the containing menu has its `item_activate_sound_path` attribute set to something that is not `None`, or `earwax.EarwaxConfig.menus.default_item_select_sound` is not `None`.

get_title() → Optional[str]

Return the proper title of this object.

If `self.title` is a callable, its return value will be returned.

on_selected() → None

Handle this menu item being selected.

class earwax.menus.**ActionMenu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, input_mode: Optional[earwax.input_modes.InputModes] = NOTHING, all_triggers_label: Optional[str] = '<< Show all triggers >>'*)

Bases: earwax.menus.menu.Menu

A menu to show a list of actions and their associated triggers.

You can use this class with any game, like so:

```
from earwax import Game, Level, ActionMenu
from pyglet.window import Window, key
w = Window(caption='Test Game')
g = Game()
l = Level()
@l.action('Show actions', symbol=key.SLASH, modifiers=key.MOD_SHIFT)
def actions_menu():
    '''Show an actions menu.'''
    a = ActionMenu(g, 'Actions')
    g.push_level(a)

g.push_level(l)
g.run(w)
```

Now, if you press shift and slash (a question mark on english keyboards), you will get an action menu.

This code can be shortened to:

```
@l.action('Show actions', symbol=key.SLASH, modifiers=key.MOD_SHIFT)
def actions_menu():
    '''Show an actions menu.'''
    game.push_action_menu()
```

If you want to override how triggers appear in the menu, then you can override symbol_to_string() and mouse_to_string().

Variables

- **input_mode** – The input mode this menu will show actions for.
- **all_triggers_label** – The label for the "All triggers" entry.

  If this value is None no such entry will be shown.

action_menu(*action: earwax.action.Action*) → Callable[[], Optional[Generator[None, None, None]]]

Show a submenu of triggers.

Override this method to change how the submenu for actions is displayed.

Parameters action – The action to generate the menu for.

action_title(*action: earwax.action.Action, triggers: List[str]*) → str

Return a suitable title for the given action.

This method is used when building the menu when input_mode is not None.

Parameters

- **action** – The action whose name will be used.

- **`triggers`** – A list of triggers gleaned from the given action.

`get_default_input_mode`() → earwax.input_modes.InputModes
Get the default input mode.

`handle_action`(*action: earwax.action.Action*) → Callable[[], Optional[Generator[None, None, None]]]
Handle an action.

This method is used as the menu handler that is triggered when you select a trigger to activate the current action.

Parameters **`action`** – The action to run.

`hat_direction_to_string`(*direction: Tuple[int, int]*) → str
Return the given hat direction as a string.

`mouse_to_string`(*action: earwax.action.Action*) → str
Describe how to trigger the given action with the mouse.

Returns a string representing the mouse button and modifiers needed to trigger the provided action.

You must be certain that `action.mouse_button is not None`.

Override this method to change how mouse triggers appear.

Parameters **`action`** – The action whose `mouse_button` attribute this method will be working on.

`show_all`() → None
Show all triggers.

`symbol_to_string`(*action: earwax.action.Action*) → str
Describe how to trigger the given action with the keyboard.

Returns a string representing the symbol and modifiers needed to trigger the provided action.

You must be certain that `action.symbol is not None`.

Override this method to change how symbol triggers appear.

Parameters **`action`** – The action whose `symbol` attribute this method will be working on.

class earwax.menus.**FileMenu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, path: pathlib.Path = NOTHING, func: Callable[[Optional[pathlib.Path]], Optional[Generator[None, None, None]]] = <built-in function print>, root: Optional[pathlib.Path] = None, empty_label: Optional[str] = None, directory_label: Optional[str] = None, show_directories: bool = True, show_files: bool = True, up_label: str = '..'*)
Bases: *`earwax.menus.menu.Menu`*

A menu for selecting a file.

File menus can be used as follows:

```
from pathlib import Path
from earwax import Game, Level, FileMenu, tts
from pyglet.window import key, Window
w = Window(caption='Test Game')
g = Game()
```

(continues on next page)

(continued from previous page)

```
l = Level(g)
@l.action('Show file menu', symbol=key.F)
def file_menu():
    '''Show a file menu.'''
    def inner(p):
        tts.speak(str(p))
        g.pop_level()
    f = FileMenu(g, 'File Menu', Path.cwd(), inner)
    g.push_level(f)

g.push_level(l)
g.run(w)
```

Variables

- **path** – The path this menu will start at.
- **func** – The function to run with the resulting file or directory.
- **root** – The root directory which this menu will be chrooted to.
- **empty_label** – The label given to an entry which will allow this menu to return None as a result.

  If this label is None (the default), then then no such option will be available.
- **directory_label** – The label given to an entry which will allow a directory - in addition to files - to be selected.

  If this argument is None (the default), then no such option will be available.

  If you only want directories to be selected, then pass show_files=False to the constructor.
- **show_directories** – Whether or not to show directories in the list.
- **show_files** – Whether or not to include files in the list.
- **up_label** – The label given to the entry to go up in the directory tree.

navigate_to(*path: pathlib.Path*) → Callable[[], None]

Navigate to a different path.

Instead of completely replacing the menu, just change the path, and re- use this instance.

rebuild_menu() → None

Rebuild the menu.

This method will be called once after initialisation, and every time the directory is changed by the `navigate_to()` method.

select_item(*path: Optional[pathlib.Path]*) → Callable[[], Optional[Generator[None, None, None]]]

Select an item.

Used as the menu handler in place of a lambda.

Parameters **path** – The path that has been selected. Could be a file or a directory.

class earwax.menus.**ConfigMenu**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, config: earwax.config.Config = NOTHING*)

Bases: `earwax.menus.menu.Menu`

A menu that allows the user to set values on configuration sections.

If an option is present with a type the menu doesn't know how to handle, `earwax.UnknownTypeError` will be raised.

Variables

- **`config`** – The configuration section this menu will configure.
- **`type_handlers`** – Functions to handle the types this menu knows about.

  New types can be handled with the `type_handler()` method.

`activate_handler`(*handler: earwax.menus.config_menu.TypeHandler*, *option: earwax.config.ConfigValue*) → Callable[[], Optional[Generator[None, None, None]]]

Activates the given handler with the given configuration value.

Used by the `option_menu()` method when building menus.

Parameters

- **`handler`** – The `TypeHandler` instance that should be activated.
- **`option`** – The `ConfigValue` instance the handler should work with.

`clear_value`(*option: earwax.config.ConfigValue*) → None

Clear the value.

Sets `option.value` to `None`.

Used by the default `TypeHandler` that handles nullable values.

Parameters **`option`** – The `ConfigValue` instance whose value should be set to `None`.

`earwax_config`() → earwax.config.Config

Return the main earwax configuration.

`get_option_name`(*option: earwax.config.ConfigValue*, *name: str*) → str

Get the name for the given option.

The provided `name` argument will be the attribute name, so should only be used if the option has no `__section_name__` attribute.

Parameters

- **`option`** – The `ConfigValue` instance whose name should be returned.
- **`name`** – The name of the attribute that holds the option.

`get_subsection_name`(*subsection: earwax.config.Config*, *name: str*) → str

Get the name for the given subsection.

The provided `name` argument will be the attribute name, so should only be used if the subsection has no `__section_name__` attribute.

Parameters

- **`subsection`** – The `Config` instance whose name should be returned.
- **`name`** – The name of the attribute that holds the subsection.

`handle_bool`(*option: earwax.config.ConfigValue*) → None

Toggle a boolean value.

Used by the default `TypeHandler` that handles boolean values.

Parameters **`option`** – The `ConfigValue` instance to work on.

`handle_float`(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow editing floats.

Used by the default `TypeHandler` that handles float values.

Parameters **`option`** – The `ConfigValue` instance to work on.

`handle_int`(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow editing integers.

Used by the default `TypeHandler` that handles integer values.

Parameters **`option`** – The `ConfigValue` instance to work on.

`handle_path`(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow selecting files and folders.

Used by the default `TypeHandler` that handles `pathlib.Path` values.

Parameters **`option`** – The `ConfigValue` instance to work on.

`handle_string`(*option: earwax.config.ConfigValue*) → Generator[None, None, None]
Allow editing strings.

Used by the default `TypeHandler` that handles string values.

Parameters **`option`** – The `ConfigValue` instance to work on.

`option_menu`(*option: earwax.config.ConfigValue*, *name: str*) → Callable[[], Generator[None, None, None]]
Add a menu for the given option.

If the type of the provided option is a `Union` type (like `Optional[str]`), then an entry for editing each type will be added to the menu. Otherwise, there will be only one entry.

The only special case is when the type is a tuple of values. If this happens, the menu will instead be populated with a list of entries corrisponding to the values of the tuple.

At the end of the menu, there will be an option to restore the default value.

Parameters

- **`option`** – The `ConfigValue` instance to generate a menu for.
- **`name`** – The proper name of the given option, as returned by `get_option_name()`.

`set_value`(*option: earwax.config.ConfigValue*, *value: Any*, *message: str = 'Done.'*) → Callable[[], None]
Set a value.

Returns a callable that can be used to set the value of the provided option to the provided value.

This method returns a callable because it is used extensively by `option_menu()`, and a bunch of lambdas becomes less readable. Plus, Mypy complains about them.

Parameters

- **`option`** – The `ConfigValue` instance to work on.
- **`value`** – The value to set `option.value` to.
- **`message`** – The message to be spoken after setting the value.

`subsection_menu`(*subsection: earwax.config.Config*, *name: str*) → Callable[[], Generator[None, None, None]]
Add a menu for the given subsection.

By default, creates a new `earwax.ConfigMenu` instance, and returns a function that - when called - will push it onto the stack.

Parameters

- **`subsection`** – The `Config` instance to create a menu for.
- **`name`** – The proper name of the subsection, returned by `get_subsection_name()`.

`type_handler`(*type_: object, title: Callable[[earwax.config.ConfigValue, str], str]*) → Callable[[Callable[[earwax.config.ConfigValue], Optional[Generator[None, None, None]]]], Callable[[earwax.config.ConfigValue], Optional[Generator[None, None, None]]]]

Add a type handler.

Decorate a function to be used as a type handler:

```
from datetime import datetime, timedelta
from earwax import ConfigMenu, tts

m = ConfigMenu(pretend_config, 'Options', game)

@m.type_handler(datetime, lambda option, name: 'Add a week')
def add_week(option):
    '''Add a week to the current value.'''
    option.value += timedelta(days=7)
    self.game.output('Added a week.')
    m.game.pop_level()
```

Handlers can do anything menu item functions can do, including creating more menus, and yielding.

Parameters

- **`type`** – The type this handler should be registered for.
- **`title`** – A function which will return the title for the menu item for this handler.

class `earwax.menus.`**`TypeHandler`**(*title: Callable[[earwax.config.ConfigValue, str], str], func: Callable[[earwax.config.ConfigValue], Optional[Generator[None, None, None]]]*)

Bases: `object`

A type handler for use with `ConfigMenu` instances.

Variables

- **`title`** – A function that will return a string which can be used as the title for the menu item generated by this handler.
- **`func`** – The function that will be called when this handler is required.

exception `earwax.menus.`**`UnknownTypeError`**

Bases: `Exception`

An unknown type was encountered.

An exception which will be thrown if a `ConfigMenu` instance doesn't know how to handle the given type.

class earwax.menus.**ReverbEditor**(*game: Game, title: Union[str, TitleFunction], dismissible: bool = True, item_select_sound_path: Optional[pathlib.Path] = None, item_activate_sound_path: Optional[pathlib.Path] = None, position: int = -1, search_timeout: float = 0.5, search_time: float = 0.0, reverb: object = NOTHING, settings: earwax.reverb.Reverb = NOTHING, setting_items: List[earwax.menus.menu_item.MenuItem] = NOTHING*)

Bases: *earwax.menus.menu.Menu*

A menu for editing reverbs.

adjust_value(*amount: earwax.menus.reverb_editor.ValueAdjustments*) → Callable[[], None]
Restore the current menu item to the default.

edit_value(*setting: earwax.menus.reverb_editor.ReverbSetting*, *value: float*) → Callable[[], Generator[None, None, None]]
Edit the given value.

get_default_reverb() → object
Raise an error.

get_default_settings() → earwax.reverb.Reverb
Raise an error.

reset() → None
Reload this menu.

set_value(*setting: earwax.menus.reverb_editor.ReverbSetting*, *value: float*) → None
Set the value.

earwax.promises package

Submodules

earwax.promises.base module

Provides the base Promise class, and the PromisesStates enumeration.

class earwax.promises.base.**Promise**

Bases: `typing.Generic`, *earwax.mixins.RegisterEventMixin*

The base class for promises.

Instances of this class have a few possible states which are contained in the `PromiseStates` enumeration.

Variables **state** – The state this promise is in (see above).

cancel() → None
Override to provide cancel functionality.

done(*value: T*) → None
Finish up.

Dispatches the `on_done()` event with the given `value`, and set `self.state` to `earwax.PromiseStates.done`.

Parameters **value** – The value that was returned from whatever function this promise had.

error(*e: Exception*) → None
Handle an error.

This event dispatches the `on_error()` event with the passed exception.

Parameters **`e`** – The exception that was raised.

`on_cancel``()` → None
Handle cancellation.

This event is dispatched when this instance has its `cancel()` method called.

`on_done``(`*result: T*`)` → None
Handle return value.

This event is dispatched when this promise completes with no error.

Parameters **`result`** – The value returned by the function.

`on_error``(`*e: Exception*`)` → None
Handle an error.

This event is dispatched when this promise raises an error.

Parameters **`e`** – The exception that was raised.

`on_finally``()` → None
Handle this promise comise completing.

This event is dispatched when this promise completes, whether or not it raises an error.

`run``(`**args*, ***kwargs*`)` → None
Start this promise running.

`class` `earwax.promises.base.`**`PromiseStates`**
Bases: `enum.Enum`

The possible states of `earwax.Promise` instances.

Variables

- **`not_ready`** – The promise has been created, but a function must still be added.

  How this is done depends on how the promise subclass in question has been implemented, and may not always be used.

- **`ready`** – The promise has been created, and a function registered. The `run()` method has not yet been called.

- **`running`** – The promise's `run()` method has been called, but the function has not yet returned a value, or raised an error.

- **`done`** – The promise has finished, and there was no error. The `on_done()` and `on_finally()` events have already been dispatched.

- **`error`** – The promise completed, but there was an error, which was handled by the `on_error()` event.

  The `on_finally()` event has been dispatched.

- **`cancelled`** – The promise has had its `cancel()` method called, and its `on_cancel()` event has been dispatched.

`cancelled = 5`

`done = 3`

`error = 4`

`not_ready = 0`

`ready = 1`

running = 2

earwax.promises.staggered_promise module

Provides the StaggeredPromise class.

class earwax.promises.staggered_promise.**StaggeredPromise**(*func: Callable[[...], Generator[float, None, T]]*)

Bases: *earwax.promises.base.Promise*

A promise that can suspend itself at will.

I found myself missing the MOO-style suspend() function, so thought I'd make the same capability available in earwax:

```
@StaggeredPromise.decorate
def promise() -> StaggeredPromiseGeneratorType:
    game.output('Hello.')
    yield 2.0
    game.output('World.')

promise.run()
game.run(window)
```

This class supports all the promise events found on `earwax.Promise`, and also has a `on_next()` event, which will fire whenever a promise suspends:

```
@promise.event
def on_next(delay: float) -> None:
    print(f'I waited {delay}.')
```

Variables

- **func** – The function to run.
- **generator** – The generator returned by `self.func`.

cancel() → None

Cancel this promise.

Cancels `self.generator`, and sets the proper state.

classmethod decorate(*func: Callable[[...], Generator[float, None, T]]*) → earwax.promises.staggered_promise.StaggeredPromise

Make an instance from a decorated function.

This function acts as a decorator method for returning `earwax.StaggeredPromise` instances.

Using this function seems to help mypy figure out what type your function is.

Parameters **func** – The function to decorate.

do_next(*dt: Optional[float]*) → None

Advance execution.

Calls `next(self.generator)`, and then suspend for however long the function demands.

If `StopIteration` is raised, then the args from that exception are sent to the `self.on_done` event.

If any other exception is raised, then that exception is passed along to the `self.on_error` event.

Parameters dt – The time since the last run, as passed by `pyglet.clock.schedule_once`.

If this is the first time this method is called, `dt` will be `None`.

on_next (*delay: float*) → None
Do something when execution is advanced.

This event is dispatched every time `next` is called on `self.func`.

Parameters delay – The delay that was requested by the function.

run (**args*, ***kwargs*) → None
Run this promise.

Start `self.func` running, and set the proper state.

Parameters

- **args** – The positional arguments passed to `self.func`.
- **kwargs** – The keyword arguments passed to `self.func`.

earwax.promises.threaded_promise module

Provides the ThreadedPromise class.

class earwax.promises.threaded_promise.**ThreadedPromise** (*thread_pool: concurrent.futures._base.Executor, func: Optional[Callable[[...], T]] = None, future: Optional[concurrent.futures._base.Future] = None*)

Bases: *earwax.promises.base.Promise*

A promise that a value will be available in the future.

Uses an `Executor` subclass (like `ThreadPoolExecutor`, or `ProcessPoolExecutor` for threading).

You can create this class directly, or by using decorators.

Here is an example of the decorator syntax:

```
from concurrent.futures import ThreadPoolExecutor

promise: ThreadedPromise = ThreadedPromise(ThreadPoolExecutor())

@promise.register_func
def func() -> None:
    # Long-running task...
    return 5

@promise.event
def on_done(value: int) -> None:
    # Do something with the return value.

@promise.event
def on_error(e: Exception) -> None:
    # Do something with an error.
```

(continues on next page)

(continued from previous page)

```
@promise.event
def on_finally():
    print('Done.')

promise.run()
```

Or you could create the promise manually:

```
promise = ThreadedPromise(
    ThreadPoolExecutor(), func=predefined_function
)
promise.event('on_done')(print)
promise.run()
```

Note the use of Pyglet's own event system.

Variables

- **`thread_pool`** – The thread pool to use.
- **`func`** – The function to submit to the thread pool.
- **`future`** – The future that is running, or None if the `run()` method has not yet been called.

`cancel``()` → None

Try to cancel `self.future`.

If There is no future, `RuntimeError` will be raised.

`check``(`*dt: float*`)` → None

Check state and react accordingly.

Checks to see if `self.future` has finished or not.

If it has, dispatch the `on_done()` event with the resulting value.

If an error has been raised, dispatch the `on_error()` event with the resulting error.

If either of these things have happened, dispatch the `on_finally()` event.

Parameters **`dt`** – The time since the last run.

This argument is required by `pyglet.clock.schedule`.

`register_func``(`*func: Callable[[...], T]*`)` → Callable[[...], T]

Register promise function.

Registers the function to be called by the `run()` method.

Parameters **`func`** – The function to use. Will be stored in `self.func`.

`run``(`**args*, ***kwargs*`)` → None

Start this promise running.

The result of calling `submit` on `self.thread_pool` will be stored on `self.future`.

If this instance does not have a function registered yet, `RuntimeError` will be raised.

Parameters

- **`args`** – The extra positional arguments to pass along to `submit`.
- **`kwargs`** – The extra keyword arguments to pass along to `submit`.

Module contents

Provides the various promise classes.

class earwax.promises.**PromiseStates**

Bases: `enum.Enum`

The possible states of `earwax.Promise` instances.

Variables

- **not_ready** – The promise has been created, but a function must still be added.

  How this is done depends on how the promise subclass in question has been implemented, and may not always be used.

- **ready** – The promise has been created, and a function registered. The `run()` method has not yet been called.
- **running** – The promise's `run()` method has been called, but the function has not yet returned a value, or raised an error.
- **done** – The promise has finished, and there was no error. The `on_done()` and `on_finally()` events have already been dispatched.
- **error** – The promise completed, but there was an error, which was handled by the `on_error()` event.

  The `on_finally()` event has been dispatched.

- **cancelled** – The promise has had its `cancel()` method called, and its `on_cancel()` event has been dispatched.

cancelled = 5

done = 3

error = 4

not_ready = 0

ready = 1

running = 2

class earwax.promises.**ThreadedPromise**(*thread_pool: concurrent.futures._base.Executor, func: Optional[Callable[[...], T]] = None, future: Optional[concurrent.futures._base.Future] = None*)

Bases: *earwax.promises.base.Promise*

A promise that a value will be available in the future.

Uses an `Executor` subclass (like `ThreadPoolExecutor`, or `ProcessPoolExecutor` for threading).

You can create this class directly, or by using decorators.

Here is an example of the decorator syntax:

```
from concurrent.futures import ThreadPoolExecutor

promise: ThreadedPromise = ThreadedPromise(ThreadPoolExecutor())

@promise.register_func
def func() -> None:
    # Long-running task...
```

(continues on next page)

(continued from previous page)

```
    return 5

@promise.event
def on_done(value: int) -> None:
    # Do something with the return value.

@promise.event
def on_error(e: Exception) -> None:
    # Do something with an error.

@promise.event
def on_finally():
    print('Done.')

promise.run()
```

Or you could create the promise manually:

```
promise = ThreadedPromise(
    ThreadPoolExecutor(), func=predefined_function
)
promise.event('on_done')(print)
promise.run()
```

Note the use of Pyglet's own event system.

Variables

- **`thread_pool`** – The thread pool to use.
- **`func`** – The function to submit to the thread pool.
- **`future`** – The future that is running, or None if the `run()` method has not yet been called.

`cancel`() → None

Try to cancel `self.future`.

If There is no future, `RuntimeError` will be raised.

`check`(*dt: float*) → None

Check state and react accordingly.

Checks to see if `self.future` has finished or not.

If it has, dispatch the `on_done()` event with the resulting value.

If an error has been raised, dispatch the `on_error()` event with the resulting error.

If either of these things have happened, dispatch the `on_finally()` event.

Parameters **`dt`** – The time since the last run.

This argument is required by `pyglet.clock.schedule`.

`register_func`(*func: Callable[[...], T]*) → Callable[[...], T]

Register promise function.

Registers the function to be called by the `run()` method.

Parameters **`func`** – The function to use. Will be stored in `self.func`.

`run`(**args*, ***kwargs*) → None

Start this promise running.

The result of calling `submit` on `self.thread_pool` will be stored on `self.future`.

If this instance does not have a function registered yet, `RuntimeError` will be raised.

Parameters

- **`args`** – The extra positional arguments to pass along to `submit`.
- **`kwargs`** – The extra keyword arguments to pass along to `submit`.

`class` `earwax.promises.`**`StaggeredPromise`**(*func: Callable[[...], Generator[float, None, T]]*)

Bases: *`earwax.promises.base.Promise`*

A promise that can suspend itself at will.

I found myself missing the MOO-style suspend() function, so thought I'd make the same capability available in earwax:

```
@StaggeredPromise.decorate
def promise() -> StaggeredPromiseGeneratorType:
    game.output('Hello.')
    yield 2.0
    game.output('World.')

promise.run()
game.run(window)
```

This class supports all the promise events found on `earwax.Promise`, and also has a `on_next()` event, which will fire whenever a promise suspends:

```
@promise.event
def on_next(delay: float) -> None:
    print(f'I waited {delay}.')
```

Variables

- **`func`** – The function to run.
- **`generator`** – The generator returned by `self.func`.

`cancel`() → None

Cancel this promise.

Cancels `self.generator`, and sets the proper state.

`classmethod decorate`(*func: Callable[[...], Generator[float, None, T]]*) → earwax.promises.staggered_promise.StaggeredPromise

Make an instance from a decorated function.

This function acts as a decorator method for returning `earwax.StaggeredPromise` instances.

Using this function seems to help mypy figure out what type your function is.

Parameters **`func`** – The function to decorate.

`do_next`(*dt: Optional[float]*) → None

Advance execution.

Calls `next(self.generator)`, and then suspend for however long the function demands.

If `StopIteration` is raised, then the args from that exception are sent to the `self.on_done` event.

If any other exception is raised, then that exception is passed along to the `self.on_error` event.

Parameters **dt** – The time since the last run, as passed by `pyglet.clock.schedule_once`.

If this is the first time this method is called, `dt` will be `None`.

on_next (*delay: float*) → None

Do something when execution is advanced.

This event is dispatched every time `next` is called on `self.func`.

Parameters **delay** – The delay that was requested by the function.

run (**args*, ***kwargs*) → None

Run this promise.

Start `self.func` running, and set the proper state.

Parameters

- **args** – The positional arguments passed to `self.func`.
- **kwargs** – The keyword arguments passed to `self.func`.

class earwax.promises.**Promise**

Bases: `typing.Generic`, *earwax.mixins.RegisterEventMixin*

The base class for promises.

Instances of this class have a few possible states which are contained in the `PromiseStates` enumeration.

Variables **state** – The state this promise is in (see above).

cancel () → None

Override to provide cancel functionality.

done (*value: T*) → None

Finish up.

Dispatches the `on_done()` event with the given `value`, and set `self.state` to `earwax.PromiseStates.done`.

Parameters **value** – The value that was returned from whatever function this promise had.

error (*e: Exception*) → None

Handle an error.

This event dispatches the `on_error()` event with the passed exception.

Parameters **e** – The exception that was raised.

on_cancel () → None

Handle cancellation.

This event is dispatched when this instance has its `cancel()` method called.

on_done (*result: T*) → None

Handle return value.

This event is dispatched when this promise completes with no error.

Parameters **result** – The value returned by the function.

on_error (*e: Exception*) → None

Handle an error.

This event is dispatched when this promise raises an error.

Parameters **e** – The exception that was raised.

on_finally() → None
Handle this promise comise completing.

This event is dispatched when this promise completes, whether or not it raises an error.

run(**args*, ***kwargs*) → None
Start this promise running.

earwax.story package

Submodules

earwax.story.context module

Provides the StoryContext class.

class earwax.story.context.**StoryContext**(*game: earwax.game.Game*, *world: earwax.story.world.StoryWorld*, *edit: bool = NOTHING*, *state: earwax.story.world.WorldState = NOTHING*, *errors: List[str] = NOTHING*, *warnings: List[str] = NOTHING*)

Bases: object

Holds references to various objects required to make a story work.

before_run() → None
Set the default panning strategy.

configure_earwax() → None
Push a menu that can be used to configure Earwax.

configure_music() → None
Allow adding and removing main menu music.

credit_menu(*credit: earwax.credit.Credit*) → Callable[[], None]
Push a menu that can deal with credits.

credits_menu() → None
Add or remove credits.

earwax_bug() → None
Open the Earwax new issue URL.

get_default_config_file() → pathlib.Path
Get the default configuration filename.

get_default_logger() → logging.Logger
Return a default logger.

get_default_state() → earwax.story.world.WorldState
Get a default state.

get_main_menu() → earwax.menus.menu.Menu
Create a main menu for this world.

get_window_caption() → str
Return a suitable window title.

load() → None
Load an existing game, and start it.

play() → None
Push the world level.

push_credits() → None
Push the credits menu.

set_initial_room() → None
Set the initial room.

set_panner_strategy() → None
Allow the changing of the panner strategy.

show_warnings() → None
Show any generated warnings.

world_options() → None
Configure the world.

earwax.story.edit_level module

Provides the EditLevel class.

class earwax.story.edit_level.**EditLevel**(*game: Game, world_context: StoryContext, cursor_sound: Optional[earwax.sound.Sound] = None, inventory: List[earwax.story.world.RoomObject] = NOTHING, reverb: Optional[GlobalFdnReverb] = None, object_ambiances: Dict[str, List[earwax.ambiance.Ambiance]] = NOTHING, object_tracks: Dict[str, List[earwax.track.Track]] = NOTHING, filename: Optional[pathlib.Path] = None, builder_menu_actions: List[earwax.action.Action] = NOTHING*)
Bases: `earwax.story.play_level.PlayLevel`

A level for editing stories.

add_action(*obj: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit, earwax.story.world.StoryWorld], name: str*) → Callable[[], None]
Add a new action to the given object.

Parameters

- **obj** – The object to assign the new action to.
- **name** – The attribute name to use.

add_ambiance(*ambiances: List[earwax.story.world.WorldAmbiance]*) → Callable[[], Generator[None, None, None]]
Add a new ambiance to the given list.

ambiance_menu(*ambiances: List[earwax.story.world.WorldAmbiance], ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], Generator[None, None, None]]
Push the edit ambiance menu.

ambiances_menu() → Generator[None, None, None]
Push a menu that can edit ambiances.

builder_menu() → Generator[None, None, None]
Push the builder menu.

configure_reverb() → None
Configure the reverb for the current room.

create_exit() → Generator[None, None, None]
Link this room to another.

create_menu() → Generator[None, None, None]
Show the creation menu.

create_object() → None
Create a new object in the current room.

create_room() → None
Create a new room.

delete() → None
Delete the currently focused object.

delete_ambiance(*ambiances: List[earwax.story.world.WorldAmbiance], ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], None]
Delete the ambiance.

describe_room() → Generator[None, None, None]
Set the description for the current room.

edit_action(*obj: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit, earwax.story.world.StoryWorld], action: earwax.story.world.WorldAction*) → Callable[[], None]
Push a menu that allows editing of the action.

Parameters

- **obj** – The object the action is attached to.
- **action** – The action to edit (or delete).

edit_ambiance(*ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], Generator[None, None, None]]
Edit the ambiance.

edit_object_class(*class_: earwax.story.world.RoomObjectClass*) → Callable[[], None]
Push a menu for editing object classes.

Parameters **class** – The object class to edit.

edit_object_class_names() → None
Push a menu that can edit object class names.

edit_object_classes() → None
Push a menu for editing object classes.

edit_volume_multiplier(*ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], Generator[None, None, None]]
Return a callable that can be used to set an ambiance volume multiplier.

Parameters **ambiance** – The ambiance whose volume multiplier will be changed.

get_rooms(*include_current: bool = True*) → List[earwax.story.world.WorldRoom]
Return a list of rooms from this world.

Parameters **include_current** – If this value is `True`, the current room will be included.

goto_room() → Generator[None, None, None]
Let the player choose a room to go to.

object_actions() → Generator[None, None, None]
Push a menu that lets you configure object actions.

remessage() → Optional[Generator[None, None, None]]
Set a message on the currently-focused object.

rename() → Generator[None, None, None]
Rename the currently focused object.

reposition_object() → None
Reposition the currently selected object.

room
Return the current room.

save_world() → None
Save the world.

set_action_sound(*action: earwax.story.world.WorldAction*) → Generator[None, None, None]
Set the sound on the given action.

Parameters **action** – The action whose sound will be changed.

set_message(*action: earwax.story.world.WorldAction*) → Generator[None, None, None]
Push an editor to set the message on the provided action.

Parameters **action** – The action whose message attribute will be modified.

set_name(*obj: Union[earwax.story.world.WorldAction, earwax.story.world.RoomObject, earwax.story.world.WorldRoom]*) → Generator[None, None, None]
Push an editor that can be used to change the name of `obj`.

Parameters **obj** – The object to rename.

set_object_type() → None
Change the type of an object.

set_world_messages() → Generator[None, None, None]
Push a menu that allows the editing of world messages.

set_world_sound(*name: str*) → Callable[[], Generator[None, None, None]]
Set the given sound.

Parameters **name** – The name of the sound to edit.

shadow_description() → None
Set the description of this room from another room.

shadow_name() → None
Sow a menu to select another room whose name will be shadowed.

sounds_menu() → Optional[Generator[None, None, None]]
Add or remove ambiances for the currently focused object.

world_sounds() → Generator[None, None, None]
Push a menu that can be used to configure world sounds.

class earwax.story.edit_level.**ObjectPositionLevel**(*game: Game, object: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit], level: EditLevel, initial_position: Optional[earwax.story.world.DumpablePoint] = NOTHING*)

Bases: `earwax.level.Level`

A level for editing the position of an object.

Variables

- **`object`** – The object or exit whose position will be edited.
- **`level`** – The edit level which pushed this level.

`backward``()` → None
Move the sound backwards.

`cancel``()` → None
Undo the move, and return everything to how it was.

`clear``()` → None
Clear the object position.

`done``()` → None
Finish editing.

`down``()` → None
Move the sound down.

`forward``()` → None
Move the sound forwards.

`get_initial_position``()` → Optional[earwax.story.world.DumpablePoint]
Get the object position.

`left``()` → None
Move the sound left.

`move``(`*x: int = 0*, *y: int = 0*, *z: int = 0*`)` → None
Change the position of this object.

`reset``()` → None
Reset the current room.

`right``()` → None
Move the sound right.

`up``()` → None
Move the sound up.

`earwax.story.edit_level.`**`push_actions_menu`**`(`*game: earwax.game.Game, actions: List[earwax.story.world.WorldAction], activate: Callable[[earwax.story.world.WorldAction], Optional[Generator[None, None, None]]]*`)` → Generator[None, None, None]

Push a menu that lets the player select an action.

Parameters

- **`game`** – The game to use when constructing the menu.
- **`actions`** – A list of actions to show.
- **`activate`** – A function to call with the chosen action.

`earwax.story.edit_level.`**`push_rooms_menu`**(*game: earwax.game.Game, rooms: List[earwax.story.world.WorldRoom], activate: Callable[[earwax.story.world.WorldRoom], Optional[Generator[None, None, None]]]*) → Generator[None, None, None]

Push a menu with all the provided rooms.

Parameters

- **`game`** – The game to pop this level from when a room is selected.
- **`rooms`** – The rooms which should show up in the menu.
- **`activate`** – The function to call with the selected room.

earwax.story.play_level module

Provides the StoryLevel class.

`class` `earwax.story.play_level.`**`PlayLevel`**(*game: Game, world_context: StoryContext, cursor_sound: Optional[earwax.sound.Sound] = None, inventory: List[earwax.story.world.RoomObject] = NOTHING, reverb: Optional[GlobalFdnReverb] = None, object_ambiances: Dict[str, List[earwax.ambiance.Ambiance]] = NOTHING, object_tracks: Dict[str, List[earwax.track.Track]] = NOTHING*)

Bases: *`earwax.level.Level`*

A level that can be used to play a story.

Instances of this class can only play stories, not edit them.

Variables

- **`world_context`** – The context that contains the world, and the state for this story.
- **`action_sounds`** – The sounds which were started by object actions.
- **`cursor_sound`** – The sound that plays when moving through objects and ambiances.
- **`inventory`** – The list of `Roomobject` instances that the player is carrying.
- **`reverb`** – The reverb object for the current room.
- **`object_ambiances`** – The ambiances for a all objects in the room, excluding those in the players' `inventory`.
- **`object_tracks`** – The tracks for each object in the current room, excluding those objects that are in the player's `inventory`.

`actions_menu`(*obj: earwax.story.world.RoomObject, menu_action: Optional[earwax.story.world.WorldAction] = None*) → None

Show a menu of object actions.

Parameters

- **`obj`** – The object which the menu will be shown for.
- **`menu_action`** – The action which will be used instead of the default `actions_action`.

activate() → None
Activate the currently focussed object.

build_inventory() → None
Build the player inventory.

This method should be performed any time *state* changes.

cycle_category(*direction: int*) → Generator[None, None, None]
Cycle through information categories.

cycle_object(*direction: int*) → None
Cycle through objects.

do_action(*action: earwax.story.world.WorldAction, obj: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit], pan: bool = True*) → None
Actually perform an action.

Parameters

- **action** – The action to perform.
- **obj** – The object that owns this action.

  If this value is of type *RoomObject*, and its `position` value is not `None`, then the action sound will be panned accordingly..
- **pan** – If this value evaluates to `False`, then regardless of the `obj` value, no panning will be performed.

drop_object(*obj: earwax.story.world.RoomObject*) → Callable[[], None]
Return a callable that can be used to drop an object.

drop_object_menu() → None
Push a menu that can be used to drop an object.

get_gain(*type: earwax.track.TrackTypes, multiplier: float*) → float
Return the proper gain.

get_objects() → List[earwax.story.world.RoomObject]
Return a list of objects that the player can see.

This method will exclude objects which are in the as yet unimplemented player inventory.

The resulting list will be sorted with Python's `sorted` builtin.

inventory_menu() → None
Show the inventory menu.

main_menu() → Generator[None, None, None]
Return to the main menu.

next_category() → Generator[None, None, None]
Next information category.

next_object() → None
Go to the next object.

object
Return the object from `self.state`.

object_menu(*obj: earwax.story.world.RoomObject*) → Callable[[], None]
Return a callable which shows the inventory menu for an object.

`objects_menu`(*objects: List[earwax.story.world.RoomObject], func: Callable[[earwax.story.world.RoomObject], Callable[[], None]], title: str*) → None
Push a menu of objects.

`on_pop`() → None
Stop all the action sounds.

`on_push`() → None
Set the initial room.

The room is the world from the *`state`* object, rather than the `initial_room`.

`pause`() → None
Pause All the currently-playing room sounds.

`perform_action`(*obj: earwax.story.world.RoomObject, action: earwax.story.world.WorldAction*) → Callable[[], None]
Return a function that will perform an object action.

This method is used by *`actions_menu()`* to allow the player to trigger object actions.

The inner method performs the following actions:

- Shows the action message to the player.
- **Plays the action sound. If `obj` has coordinates, the sound will be** heard at those coordinates.
- Pops the level to remove the actions menu from the stack.

Parameters

- **`obj`** – The object which has the action.
- **`action`** – The action which should be performed.

`play_action_sound`(*sound: str, position: Optional[earwax.point.Point] = None*) → None
Play an action sound.

Parameters

- **`sound`** – The filename of the sound to play.
- **`position`** – The position of the owning object.

  If this value is `None`, the sound will not be panned.

`play_cursor_sound`(*position: Optional[earwax.point.Point]*) → None
Play and set the cursor sound.

`play_object_ambiances`(*obj: earwax.story.world.RoomObject*) → None
Play all the ambiances for the given object.

Parameters **`obj`** – The object whose ambiances will be played.

`previous_category`() → Generator[None, None, None]
Previous information category.

`previous_object`() → None
Go to the previous object.

`save_state`() → None
Save the current state.

`set_room`(*room: earwax.story.world.WorldRoom*) → None
Move to a new room.

state
Return the current state.

stop_action_sounds() → None
Stop all action sounds.

stop_object_ambiances(*obj: earwax.story.world.RoomObject*) → None
Stop all the ambiances for the given object.

Parameters **obj** – The object whose ambiances will be stopped.

take_object(*obj: earwax.story.world.RoomObject*) → None
Take an object.

use_exit(*x: earwax.story.world.RoomExit*) → None
Use the given exit.

This method is called by the `activate()` method.

Parameters **x** – The exit to use.

use_object(*obj: earwax.story.world.RoomObject*) → Callable[[], None]
Return a callable that can be used to use an object.

use_object_menu() → None
Push a menu that allows using an object.

world
Get the attached world.

earwax.story.world module

Provides various classes relating to worlds.

class earwax.story.world.**DumpablePoint**(*x: T, y: T, z: T*)
Bases: `earwax.point.Point`, `earwax.mixins.DumpLoadMixin`

A point that can be dumped and loaded.

class earwax.story.world.**DumpableReverb**(*gain: float = 1.0, late_reflections_delay: float = 0.01, late_reflections_diffusion: float = 1.0, late_reflections_hf_reference: float = 500.0, late_reflections_hf_rolloff: float = 0.5, late_reflections_lf_reference: float = 200.0, late_reflections_lf_rolloff: float = 1.0, late_reflections_modulation_depth: float = 0.01, late_reflections_modulation_frequency: float = 0.5, mean_free_path: float = 0.02, t60: float = 1.0*)
Bases: `earwax.reverb.Reverb`, `earwax.mixins.DumpLoadMixin`

A reverb that can be dumped.

class earwax.story.world.**RoomExit**(*destination_id: str, action: earwax.story.world.WorldAction = NOTHING, position: Optional[earwax.story.world.DumpablePoint] = None*)
Bases: `earwax.mixins.DumpLoadMixin`

An exit between two rooms.

Instances of this class rely on their `action` property to show messages and play sounds, as well as for the name of the exit.

The actual destination can be retrieved with the *destination* property.

Variables

- **destination_id** – The ID of the room on the other side of this exit.
- **location** – The location of this exit.

  This value is provided by the containing *StoryWorld* class.
- **action** – An action to perform when using this exit.
- **position** – The position of this exit.

  If this value is None, then any ambiances will not be panned.

destination

Return the room this exit leads from.

This value is inferred from destination_id.

class earwax.story.world.**RoomObject**(*id: str = NOTHING*, *name: str = 'Unnamed Object'*, *actions_action: Optional[earwax.story.world.WorldAction] = None*, *ambiances: List[earwax.story.world.WorldAmbiance] = NOTHING*, *actions: List[earwax.story.world.WorldAction] = NOTHING*, *position: Optional[earwax.story.world.DumpablePoint] = None*, *drop_action: Optional[earwax.story.world.WorldAction] = None*, *take_action: Optional[earwax.story.world.WorldAction] = None*, *use_action: Optional[earwax.story.world.WorldAction] = None*, *type: earwax.story.world.RoomObjectTypes = NOTHING*, *class_names: List[str] = NOTHING*)

Bases: *earwax.story.world.StringMixin*, *earwax.mixins.DumpLoadMixin*

An object in the story.

Instances of this class will either sit in a room, or be in the player's inventory.

Variables

- **id** – The unique ID of this object. If this ID is not provided, then picking it up will not be reliable, as the ID will be randomly generated.

  Other than the above restriction, you can set the ID to be whatever you like.
- **name** – The name of this object.

  This value will be used in any list of objects.
- **actions_action** – An action object which will be used when viewing the actions menu for this object.

  If this value is None, no music will play when viewing the actions menu for this object, and the actions_menu message will be shown.
- **ambiances** – A list of ambiances to play at the position of this object.
- **actions** – A list of actions that can be performed on this object.
- **position** – The position of this object.

  If this value is None, then any ambiances will not be panned.

- **drop_action** – The action that will be used when this object is dropped by the player.

  If this value is `None`, the containing world's `drop_action` attribute will be used.
- **take_action** – The action that will be used when this object is taken by the player.

  If this value is `None`, the containing world's `take_action` attribute will be used.
- **use_action** – The action that will be used when this object is used by the player.

  If this value is `None`, then this object is considered unusable.
- **type** – Specifies what sort of object this is.
- **class_names** – The names of all the classes this object belongs to.

  If you want a list of *RoomObjectClass* instances, use the *classes* property.
- **location** – The room where this object is located.

  This value is set by the *StoryWorld()* which holds this instance.

  If this object is picked up, the location will not change, but this object will be removed from the location's `objects` dictionary.

classes

Return a list of classes.

This value is inferred from the `class_names` list.

is_droppable

Return `True` if this object can be dropped.

is_stuck

Return `True` if this object is stuck.

is_takeable

Return `True` if this object can be taken.

is_usable

Return `True` if this object can be used.

class earwax.story.world.**RoomObjectClass**(*name: str*)

Bases: *earwax.mixins.DumpLoadMixin*

Add a class for objects.

Instances of this class let you organise objects into classes.

This is used for making exits discriminate.

Variables **name** – The name of the class.

class earwax.story.world.**RoomObjectTypes**

Bases: `enum.Enum`

The type of a room object.

Variables

- ***stuck*** – This object cannot be moved.
- ***takeable*** – This object can be picked up.
- ***droppable*** – This object can be dropped.

  This value automatically implies *takeable*.

droppable = 2

stuck = 0

takeable = 1

usable = 4

class earwax.story.world.**StoryWorld**(*game: Game*, *name: str = 'Untitled World'*, *author: str = 'Unknown'*, *main_menu_musics: List[str] = NOTHING*, *cursor_sound: Optional[str] = None*, *empty_category_sound: Optional[str] = None*, *end_of_category_sound: Optional[str] = None*, *rooms: Dict[str, earwax.story.world.WorldRoom] = NOTHING*, *initial_room_id: Optional[str] = None*, *messages: earwax.story.world.WorldMessages = NOTHING*, *take_action: earwax.story.world.WorldAction = NOTHING*, *drop_action: earwax.story.world.WorldAction = NOTHING*, *panner_strategy: str = NOTHING*, *object_classes: List[earwax.story.world.RoomObjectClass] = NOTHING*)

Bases: `earwax.mixins.DumpLoadMixin`

The top level world object.

Worlds can contain rooms and messages, as well as various pieces of information about themselves.

Variables

- **game** – The game this world is part of.
- **name** – The name of this world.
- **author** – The author of this world.

  The format of this value is arbitrary, although `Author Name <author@domain.com>` is recommended.
- **main_menu_musics** – A list of filenames to play as music while the main menu is being shown.
- **cursor_sound** – The sound that will play when moving over objects.

  If this value is `None`, no sound will be heard.
- **empty_category_sound** – The sound which will be heard when cycling to an empty category.
- **end_of_category_sound** – The sound which will be heard when cycling to the end of a category.
- **rooms** – A mapping of room IDs to rooms.
- **initial_room_id** – The ID of the room to be used when first starting the game.
- **messages** – The messages object used by this world.
- **take_action** – The default take action.

  This value will be used when an object is taken with its `take_action` attribute set to `None`.
- **drop_action** – The default drop action.

  This value will be used when an object is dropped and has its `drop_action` attribute is `None`.
- **panner_strategy** – The name of the default `panner strategy` to use.

- **object_classes** – A list of object classes.

  Objects are mapped to these classes by way of their `class_names` and *classes* lists.

add_room(*room: earwax.story.world.WorldRoom*, *initial: Optional[bool] = None*) → None
Add a room to this world.

Parameters

- **room** – The room to add.
- **initial** – An optional boolean to specify whether the given room should become the *initial_room* or not.

  If this value is `None`, then this room will be set as default if `initial_room_id` is itself `None`.

all_objects() → Iterator[earwax.story.world.RoomObject]
Return a generator of every object contained by this world.

dump() → Dict[str, Any]
Dump this world.

initial_room
Return the initial room for this world.

classmethod load(*data: Dict[str, Any]*, **args*) → Any
Load credits before anything else.

class earwax.story.world.**StringMixin**
Bases: `object`

Provides an `__str__` method.

class earwax.story.world.**WorldAction**(*name: str = 'Unnamed Action'*, *message: Optional[str] = None*, *sound: Optional[str] = None*, *rumble_value: float = 0.0*, *rumble_duration: int = 0*)
Bases: *earwax.mixins.DumpLoadMixin*

An action that can be performed.

Actions are used by the *RoomObject* and *RoomExit* classes.

If attached to a *RoomObject* instance, its `name` will appear in the `action menu`. If attached to a *RoomExit* instance, then its `name` will appear in the exits list.

Variables

- **name** – The name of this action.
- **message** – The message that is shown to the player when this action is used.

  If this value is omitted, no message will be shown.
- **sound** – The sound that should play when this action is used.

  If this value is omitted, no sound will be heard.
- **rumble_value** – The power of a rumble triggered by this action.

  This value should be between 0.0 (nothing) and 1.0 (full power).

  If this value is `0`, no rumble will occur.
- **rumble_duration** – The time (in seconds) the rumble should continue for.

  If this value is `0`, no rumble will occur.

class earwax.story.world.**WorldAmbiance**(*path: str*, *volume_multiplier: float = 1.0*)

Bases: *earwax.mixins.DumpLoadMixin*

An ambiance.

This class represents a looping sound, which is either attached to a *WorldRoom* instance, or a *RoomObject* instance.

Variables

- **path** – The path to a sound file.
- **volume_multiplier** – A value to multiply the ambiance volume by to get the volume for this sound..

class earwax.story.world.**WorldMessages**(*no_objects: str = 'This room is empty.'*, *no_actions: str = 'There is nothing you can do with this object.'*, *no_exits: str = 'There is no way out of this room.'*, *no_use: str = 'You cannot use {}.'*, *nothing_to_use: str = 'You have nothing that can be used.'*, *nothing_to_drop: str = 'You have nothing that can be dropped.'*, *empty_inventory: str = "You aren't carrying anything."*, *room_activate: str = 'You cannot do that.'*, *room_category: str = 'Location'*, *objects_category: str = 'Objects'*, *exits_category: str = 'Exits'*, *actions_menu: str = 'You step up to {}.'*, *inventory_menu: str = 'Inventory'*, *main_menu: str = 'Main Menu'*, *play_game: str = 'Start new game'*, *load_game: str = 'Load game'*, *show_credits: str = 'Show Credits'*, *credits_menu: str = 'Credits'*, *welcome: str = 'Welcome to this game.'*, *no_saved_game: str = 'You have no game saved.'*, *exit: str = 'Exit'*)

Bases: *earwax.mixins.DumpLoadMixin*

All the messages that can be shown to the player.

When adding a message to this class, make sure to add the same message and an appropriate description to the `message_descriptions` in `earwax/story/edit_level.py`.

Variables

- **no_objects** – The message which is shown when the player cycles to an empty list of objects.
- **no_actions** – The message which is shown when there are no actions for an object.
- **no_exits** – The message which is shown when the player cycles to an empty list of exits.
- **no_use** – The message which is shown when the player tries to use an object which cannot be used.
- **nothing_to_use** – The message which is shown when accessing the use menu with no usable objects.
- **nothing_to_drop** – The message which is shown when accessing the drop menu with no droppable items.
- **empty_inventory** – The message which is shown when trying to access an empty inventory menu.
- **room_activate** – The message which is shown when enter is pressed with the room category selected.

Maybe an action attribute should be added to rooms, so that enter can be used everywhere?

- **room_category** – The name of the "room" category.
- **objects_category** – The name of the "objects" category.
- **exits_category** – The name of the "exits" category.
- **actions_menu** – The message which is shown when the actions menu is activated.
- **inventory_menu** – The title of the inventory menu.

  You can include the name of the object in question, by including a set of braces:

  ```
  <message id="actions_menu">You examine {}.</message>
  ```

- **main_menu** – The title of the main menu.
- **play_game** – The title of the "play game" entry in the main menu.
- **load_game** – The title of the "load game" entry in the main menu.
- **show_credits** – The title of the "show credits" entry in the main menu.
- **credits_menu** – The title of the credits menu.
- **welcome** – The message which is shown when play starts.
- **no_saved_game** – The message which is spoken when there is no game to load.
- **exit** – The title of the "exit" entry of the main menu.

class earwax.story.world.**WorldRoom**(*id: str = NOTHING, name: str = 'Unnamed Room', description: str = 'Not described.', ambiances: List[earwax.story.world.WorldAmbiance] = NOTHING, objects: Dict[str, earwax.story.world.RoomObject] = NOTHING, exits: List[earwax.story.world.RoomExit] = NOTHING, reverb: Optional[earwax.story.world.DumpableReverb] = None*)

Bases: *earwax.mixins.DumpLoadMixin*, *earwax.story.world.StringMixin*

A room in a world.

Rooms can contain exits and object.

It is worth noting that both the room `name` and `description` can either be straight text, or they can consist of a hash character (#) followed by the ID of another room, from which the relevant attribute will be presented at runtime.

If this is the case, changing the name or description of the referenced room will change the corresponding attribute on the first instance.

This convertion can only happen once, as otherwise there is a risk of circular dependencies, causing a `RecursionError` to be raised.

Variables

- **world** – The world this room is part of.

  This value is set by the containing `StoryRoom` instance.

- **id** – The unique ID of this room.

  If this value is not provided, then an ID will be generated, based on the number of rooms that have already been loaded.

If you want to link this room with exits, it is *highly* recommended that you provide your own ID.

- **`name`** – The name of this room, or the #id of a room to inherit the name from.
- **`description`** – The description of this room, or the #id of another room to inherit the description from.
- **`ambiances`** – A list of ambiances to play when this room is in focus.
- **`objects`** – A mapping of object ids to objects.

  To get a list of objects, the canonical way is to use the *`earwax.story.play_level.PlayLevel.get_objects()`* method, as this will properly hide objects which are in the player's inventory.
- **`exits`** – A list of exits from this room.

`create_exit`(*destination: earwax.story.world.WorldRoom*, ***kwargs*) → earwax.story.world.RoomExit

Create and return an exit that links this room to another.

This method will add the new exits to this room's `exits` list, and set the appropriate `location` on the new exit.

Parameters

- **`destination`** – The destination whose ID will become the new exit's `destination_id`.
- **`kwargs`** – Extra keyword arguments to pass to the *`RoomExit`* constructor..

`create_object`(***kwargs*) → earwax.story.world.RoomObject

Create and return an exit from the provided `kwargs`.

This method will add the created object to this room's `objects` dictionary, and set the appropriate `location` attribute.

Parameters **`kwargs`** – Keyword arguments to pass to the constructor of *`RoomObject`*.

`get_description`() → str

Return the actual description of this room.

`get_name`() → str

Return the actual name of this room.

`class` `earwax.story.world.`**`WorldState`**(*world: earwax.story.world.StoryWorld*, *room_id: str = NOTHING*, *inventory_ids: List[str] = NOTHING*, *category_index: int = NOTHING*, *object_index: Optional[int] = None*)

Bases: *`earwax.mixins.DumpLoadMixin`*

The state of a story.

With the exception of the `world` attribute, this class should only have primitive types as its attributes, so that instances can be easily dumped to yaml.

Variables

- **`world`** – The world this state represents.
- **`room_id`** – The ID of the current room.
- **`inventory_ids`** – A list of object IDs which make up the player's inventory.
- **`category_index`** – The player's position in the list of categories.

- **`object_index`** – The player's position in the current category.

`category`
Return the current category.

`get_default_room_id``()` → str
Get the first room ID from the attached world.

Parameters **`instance`** – The instance to work on.

`room`
Get the current room.

class `earwax.story.world.`**`WorldStateCategories`**
Bases: `enum.Enum`

The various categories the player can select.

Variables

- ***`room`*** – The category where the name and description of a room are shown.
- ***`objects`*** – The category where the objects of a room are shown.
- ***`exits`*** – The category where the exits of a room are shown.

`exits = 2`

`objects = 1`

`room = 0`

Module contents

The story module.

Stories are a way of building worlds with no code at all.

They can do a fair amount on their own: You can create rooms, exits, objects, and you can add basic actions to those objects. In addition, you can create complex actions if you code them in yourself.

What you get out of the box:

- An easy way of creating worlds with an on screen editor.
- **A main menu, with items for playing, exiting, showing credits, and loading** saved games.
- Basic keyboard and controller commands for interracting with your world.
- **The ability to create rich 3d environments, with all the sounds, messages,** and music you can think of.
- **The ability to build your world into a single Python file you can compile** with a tool such as PyInstaller, or send about as is.

If you do wish to extend your world, build it into a Python file, then edit it to add extra actions, tasks, or whatever else you can think of.

class `earwax.story.`**`DumpablePoint`**`(`*x: T*, *y: T*, *z: T*`)`
Bases: *`earwax.point.Point`*, *`earwax.mixins.DumpLoadMixin`*

A point that can be dumped and loaded.

class earwax.story.**DumpableReverb**(*gain: float = 1.0*, *late_reflections_delay: float = 0.01*, *late_reflections_diffusion: float = 1.0*, *late_reflections_hf_reference: float = 500.0*, *late_reflections_hf_rolloff: float = 0.5*, *late_reflections_lf_reference: float = 200.0*, *late_reflections_lf_rolloff: float = 1.0*, *late_reflections_modulation_depth: float = 0.01*, *late_reflections_modulation_frequency: float = 0.5*, *mean_free_path: float = 0.02*, *t60: float = 1.0*)

Bases: `earwax.reverb.Reverb`, `earwax.mixins.DumpLoadMixin`

A reverb that can be dumped.

class earwax.story.**RoomExit**(*destination_id: str*, *action: earwax.story.world.WorldAction = NOTHING*, *position: Optional[earwax.story.world.DumpablePoint] = None*)

Bases: `earwax.mixins.DumpLoadMixin`

An exit between two rooms.

Instances of this class rely on their `action` property to show messages and play sounds, as well as for the name of the exit.

The actual destination can be retrieved with the `destination` property.

Variables

- **destination_id** – The ID of the room on the other side of this exit.
- **location** – The location of this exit.

  This value is provided by the containing `StoryWorld` class.
- **action** – An action to perform when using this exit.
- **position** – The position of this exit.

  If this value is `None`, then any `ambiances` will not be panned.

destination

Return the room this exit leads from.

This value is inferred from `destination_id`.

class earwax.story.**RoomObject**(*id: str = NOTHING*, *name: str = 'Unnamed Object'*, *actions_action: Optional[earwax.story.world.WorldAction] = None*, *ambiances: List[earwax.story.world.WorldAmbiance] = NOTHING*, *actions: List[earwax.story.world.WorldAction] = NOTHING*, *position: Optional[earwax.story.world.DumpablePoint] = None*, *drop_action: Optional[earwax.story.world.WorldAction] = None*, *take_action: Optional[earwax.story.world.WorldAction] = None*, *use_action: Optional[earwax.story.world.WorldAction] = None*, *type: earwax.story.world.RoomObjectTypes = NOTHING*, *class_names: List[str] = NOTHING*)

Bases: `earwax.story.world.StringMixin`, `earwax.mixins.DumpLoadMixin`

An object in the story.

Instances of this class will either sit in a room, or be in the player's inventory.

Variables

- **id** – The unique ID of this object. If this ID is not provided, then picking it up will not be reliable, as the ID will be randomly generated.

Other than the above restriction, you can set the ID to be whatever you like.

- **`name`** – The name of this object.

  This value will be used in any list of objects.

- **`actions_action`** – An action object which will be used when viewing the actions menu for this object.

  If this value is `None`, no music will play when viewing the actions menu for this object, and the `actions_menu` message will be shown.

- **`ambiances`** – A list of ambiances to play at the `position` of this object.
- **`actions`** – A list of actions that can be performed on this object.
- **`position`** – The position of this object.

  If this value is `None`, then any `ambiances` will not be panned.

- **`drop_action`** – The action that will be used when this object is dropped by the player.

  If this value is `None`, the containing world's `drop_action` attribute will be used.

- **`take_action`** – The action that will be used when this object is taken by the player.

  If this value is `None`, the containing world's `take_action` attribute will be used.

- **`use_action`** – The action that will be used when this object is used by the player.

  If this value is `None`, then this object is considered unusable.

- **`type`** – Specifies what sort of object this is.
- **`class_names`** – The names of all the classes this object belongs to.

  If you want a list of *`RoomObjectClass`* instances, use the *`classes`* property.

- **`location`** – The room where this object is located.

  This value is set by the *`StoryWorld()`* which holds this instance.

  If this object is picked up, the location will not change, but this object will be removed from the location's `objects` dictionary.

`classes`

Return a list of classes.

This value is inferred from the `class_names` list.

`is_droppable`

Return `True` if this object can be dropped.

`is_stuck`

Return `True` if this object is stuck.

`is_takeable`

Return `True` if this object can be taken.

`is_usable`

Return `True` if this object can be used.

class `earwax.story.`**`RoomObjectClass`**(*name: str*)

Bases: *`earwax.mixins.DumpLoadMixin`*

Add a class for objects.

Instances of this class let you organise objects into classes.

This is used for making exits discriminate.

Variables **`name`** – The name of the class.

class `earwax.story.`**`RoomObjectTypes`**

Bases: `enum.Enum`

The type of a room object.

Variables

- ***`stuck`*** – This object cannot be moved.
- ***`takeable`*** – This object can be picked up.
- ***`droppable`*** – This object can be dropped.

  This value automatically implies *`takeable`*.

`droppable = 2`

`stuck = 0`

`takeable = 1`

`usable = 4`

class `earwax.story.`**`StoryWorld`**(*game: Game, name: str = 'Untitled World', author: str = 'Unknown', main_menu_musics: List[str] = NOTHING, cursor_sound: Optional[str] = None, empty_category_sound: Optional[str] = None, end_of_category_sound: Optional[str] = None, rooms: Dict[str, earwax.story.world.WorldRoom] = NOTHING, initial_room_id: Optional[str] = None, messages: earwax.story.world.WorldMessages = NOTHING, take_action: earwax.story.world.WorldAction = NOTHING, drop_action: earwax.story.world.WorldAction = NOTHING, panner_strategy: str = NOTHING, object_classes: List[earwax.story.world.RoomObjectClass] = NOTHING*)

Bases: *`earwax.mixins.DumpLoadMixin`*

The top level world object.

Worlds can contain rooms and messages, as well as various pieces of information about themselves.

Variables

- **`game`** – The game this world is part of.
- **`name`** – The name of this world.
- **`author`** – The author of this world.

  The format of this value is arbitrary, although `Author Name <author@domain.com>` is recommended.
- **`main_menu_musics`** – A list of filenames to play as music while the main menu is being shown.
- **`cursor_sound`** – The sound that will play when moving over objects.

  If this value is `None`, no sound will be heard.
- **`empty_category_sound`** – The sound which will be heard when cycling to an empty category.
- **`end_of_category_sound`** – The sound which will be heard when cycling to the end of a category.

- **rooms** – A mapping of room IDs to rooms.
- **initial_room_id** – The ID of the room to be used when first starting the game.
- **messages** – The messages object used by this world.
- **take_action** – The default take action.

  This value will be used when an object is taken with its `take_action` attribute set to `None`.
- **drop_action** – The default drop action.

  This value will be used when an object is dropped and has its `drop_action` attribute is `None`.
- **panner_strategy** – The name of the default `panner strategy` to use.
- **object_classes** – A list of object classes.

  Objects are mapped to these classes by way of their `class_names` and *`classes`* lists.

add_room(*room: earwax.story.world.WorldRoom*, *initial: Optional[bool] = None*) → None

Add a room to this world.

Parameters

- **room** – The room to add.
- **initial** – An optional boolean to specify whether the given room should become the *`initial_room`* or not.

  If this value is `None`, then this room will be set as default if `initial_room_id` is itself `None`.

all_objects() → Iterator[earwax.story.world.RoomObject]

Return a generator of every object contained by this world.

dump() → Dict[str, Any]

Dump this world.

initial_room

Return the initial room for this world.

classmethod load(*data: Dict[str, Any]*, **args*) → Any

Load credits before anything else.

class earwax.story.**WorldAction**(*name: str = 'Unnamed Action'*, *message: Optional[str] = None*, *sound: Optional[str] = None*, *rumble_value: float = 0.0*, *rumble_duration: int = 0*)

Bases: *`earwax.mixins.DumpLoadMixin`*

An action that can be performed.

Actions are used by the *`RoomObject`* and *`RoomExit`* classes.

If attached to a *`RoomObject`* instance, its `name` will appear in the `action menu`. If attached to a *`RoomExit`* instance, then its `name` will appear in the exits list.

Variables

- **name** – The name of this action.
- **message** – The message that is shown to the player when this action is used.

  If this value is omitted, no message will be shown.

- **sound** – The sound that should play when this action is used.

  If this value is omitted, no sound will be heard.

- **rumble_value** – The power of a rumble triggered by this action.

  This value should be between 0.0 (nothing) and 1.0 (full power).

  If this value is `0`, no rumble will occur.

- **rumble_duration** – The time (in seconds) the rumble should continue for.

  If this value is `0`, no rumble will occur.

class `earwax.story.`**WorldAmbiance**(*path: str*, *volume_multiplier: float = 1.0*)

Bases: *earwax.mixins.DumpLoadMixin*

An ambiance.

This class represents a looping sound, which is either attached to a *WorldRoom* instance, or a *RoomObject* instance.

Variables

- **path** – The path to a sound file.
- **volume_multiplier** – A value to multiply the ambiance volume by to get the volume for this sound..

class `earwax.story.`**WorldMessages**(*no_objects: str = 'This room is empty.'*, *no_actions: str = 'There is nothing you can do with this object.'*, *no_exits: str = 'There is no way out of this room.'*, *no_use: str = 'You cannot use {}.'*, *nothing_to_use: str = 'You have nothing that can be used.'*, *nothing_to_drop: str = 'You have nothing that can be dropped.'*, *empty_inventory: str = "You aren't carrying anything."*, *room_activate: str = 'You cannot do that.'*, *room_category: str = 'Location'*, *objects_category: str = 'Objects'*, *exits_category: str = 'Exits'*, *actions_menu: str = 'You step up to {}.'*, *inventory_menu: str = 'Inventory'*, *main_menu: str = 'Main Menu'*, *play_game: str = 'Start new game'*, *load_game: str = 'Load game'*, *show_credits: str = 'Show Credits'*, *credits_menu: str = 'Credits'*, *welcome: str = 'Welcome to this game.'*, *no_saved_game: str = 'You have no game saved.'*, *exit: str = 'Exit'*)

Bases: *earwax.mixins.DumpLoadMixin*

All the messages that can be shown to the player.

When adding a message to this class, make sure to add the same message and an appropriate description to the `message_descriptions` in `earwax/story/edit_level.py`.

Variables

- **no_objects** – The message which is shown when the player cycles to an empty list of objects.
- **no_actions** – The message which is shown when there are no actions for an object.
- **no_exits** – The message which is shown when the player cycles to an empty list of exits.
- **no_use** – The message which is shown when the player tries to use an object which cannot be used.
- **nothing_to_use** – The message which is shown when accessing the use menu with no usable objects.

- **nothing_to_drop** – The message which is shown when accessing the drop menu with no droppable items.
- **empty_inventory** – The message which is shown when trying to access an empty inventory menu.
- **room_activate** – The message which is shown when enter is pressed with the room category selected.

  Maybe an action attribute should be added to rooms, so that enter can be used everywhere?
- **room_category** – The name of the "room" category.
- **objects_category** – The name of the "objects" category.
- **exits_category** – The name of the "exits" category.
- **actions_menu** – The message which is shown when the actions menu is activated.
- **inventory_menu** – The title of the inventory menu.

  You can include the name of the object in question, by including a set of braces:

  ```
  <message id="actions_menu">You examine {}.</message>
  ```
- **main_menu** – The title of the main menu.
- **play_game** – The title of the "play game" entry in the main menu.
- **load_game** – The title of the "load game" entry in the main menu.
- **show_credits** – The title of the "show credits" entry in the main menu.
- **credits_menu** – The title of the credits menu.
- **welcome** – The message which is shown when play starts.
- **no_saved_game** – The message which is spoken when there is no game to load.
- **exit** – The title of the "exit" entry of the main menu.

class earwax.story.**WorldRoom**(*id: str = NOTHING, name: str = 'Unnamed Room', description: str = 'Not described.', ambiances: List[earwax.story.world.WorldAmbiance] = NOTHING, objects: Dict[str, earwax.story.world.RoomObject] = NOTHING, exits: List[earwax.story.world.RoomExit] = NOTHING, reverb: Optional[earwax.story.world.DumpableReverb] = None*)

Bases: `earwax.mixins.DumpLoadMixin`, `earwax.story.world.StringMixin`

A room in a world.

Rooms can contain exits and object.

It is worth noting that both the room `name` and `description` can either be straight text, or they can consist of a hash character (#) followed by the ID of another room, from which the relevant attribute will be presented at runtime.

If this is the case, changing the name or description of the referenced room will change the corresponding attribute on the first instance.

This convertion can only happen once, as otherwise there is a risk of circular dependencies, causing a `RecursionError` to be raised.

Variables

- **`world`** – The world this room is part of.

  This value is set by the containing `StoryRoom` instance.

- **`id`** – The unique ID of this room.

  If this value is not provided, then an ID will be generated, based on the number of rooms that have already been loaded.

  If you want to link this room with exits, it is *highly* recommended that you provide your own ID.

- **`name`** – The name of this room, or the #id of a room to inherit the name from.
- **`description`** – The description of this room, or the #id of another room to inherit the description from.
- **`ambiances`** – A list of ambiances to play when this room is in focus.
- **`objects`** – A mapping of object ids to objects.

  To get a list of objects, the canonical way is to use the `earwax.story.play_level.PlayLevel.get_objects()` method, as this will properly hide objects which are in the player's inventory.

- **`exits`** – A list of exits from this room.

`create_exit`(*destination: earwax.story.world.WorldRoom*, ***kwargs*) → earwax.story.world.RoomExit

Create and return an exit that links this room to another.

This method will add the new exits to this room's `exits` list, and set the appropriate `location` on the new exit.

Parameters

- **`destination`** – The destination whose ID will become the new exit's `destination_id`.
- **`kwargs`** – Extra keyword arguments to pass to the `RoomExit` constructor..

`create_object`(***kwargs*) → earwax.story.world.RoomObject

Create and return an exit from the provided `kwargs`.

This method will add the created object to this room's `objects` dictionary, and set the appropriate `location` attribute.

Parameters **`kwargs`** – Keyword arguments to pass to the constructor of `RoomObject`.

`get_description`() → str

Return the actual description of this room.

`get_name`() → str

Return the actual name of this room.

class `earwax.story.`**`WorldState`**(*world: earwax.story.world.StoryWorld*, *room_id: str = NOTHING*, *inventory_ids: List[str] = NOTHING*, *category_index: int = NOTHING*, *object_index: Optional[int] = None*)

Bases: `earwax.mixins.DumpLoadMixin`

The state of a story.

With the exception of the `world` attribute, this class should only have primitive types as its attributes, so that instances can be easily dumped to yaml.

Variables

- **world** – The world this state represents.
- **room_id** – The ID of the current room.
- **inventory_ids** – A list of object IDs which make up the player's inventory.
- **category_index** – The player's position in the list of categories.
- **object_index** – The player's position in the current category.

category
Return the current category.

get_default_room_id() → str
Get the first room ID from the attached world.

Parameters **instance** – The instance to work on.

room
Get the current room.

class earwax.story.**WorldStateCategories**
Bases: enum.Enum

The various categories the player can select.

Variables

- ***room*** – The category where the name and description of a room are shown.
- ***objects*** – The category where the objects of a room are shown.
- ***exits*** – The category where the exits of a room are shown.

exits = 2

objects = 1

room = 0

class earwax.story.**EditLevel**(*game: Game*, *world_context: StoryContext*, *cursor_sound: Optional[earwax.sound.Sound] = None*, *inventory: List[earwax.story.world.RoomObject] = NOTHING*, *reverb: Optional[GlobalFdnReverb] = None*, *object_ambiances: Dict[str, List[earwax.ambiance.Ambiance]] = NOTHING*, *object_tracks: Dict[str, List[earwax.track.Track]] = NOTHING*, *filename: Optional[pathlib.Path] = None*, *builder_menu_actions: List[earwax.action.Action] = NOTHING*)
Bases: *earwax.story.play_level.PlayLevel*

A level for editing stories.

add_action(*obj: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit, earwax.story.world.StoryWorld], name: str*) → Callable[[], None]
Add a new action to the given object.

Parameters

- **obj** – The object to assign the new action to.
- **name** – The attribute name to use.

add_ambiance(*ambiances: List[earwax.story.world.WorldAmbiance]*) → Callable[[], Generator[None, None, None]]
Add a new ambiance to the given list.

ambiance_menu(*ambiances: List[earwax.story.world.WorldAmbiance], ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], Generator[None, None, None]]
Push the edit ambiance menu.

ambiances_menu() → Generator[None, None, None]
Push a menu that can edit ambiances.

builder_menu() → Generator[None, None, None]
Push the builder menu.

configure_reverb() → None
Configure the reverb for the current room.

create_exit() → Generator[None, None, None]
Link this room to another.

create_menu() → Generator[None, None, None]
Show the creation menu.

create_object() → None
Create a new object in the current room.

create_room() → None
Create a new room.

delete() → None
Delete the currently focused object.

delete_ambiance(*ambiances: List[earwax.story.world.WorldAmbiance], ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], None]
Delete the ambiance.

describe_room() → Generator[None, None, None]
Set the description for the current room.

edit_action(*obj: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit, earwax.story.world.StoryWorld], action: earwax.story.world.WorldAction*) → Callable[[], None]
Push a menu that allows editing of the action.

Parameters

- **obj** – The object the action is attached to.
- **action** – The action to edit (or delete).

edit_ambiance(*ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], Generator[None, None, None]]
Edit the ambiance.

edit_object_class(*class_: earwax.story.world.RoomObjectClass*) → Callable[[], None]
Push a menu for editing object classes.

Parameters class – The object class to edit.

edit_object_class_names() → None
Push a menu that can edit object class names.

edit_object_classes() → None
Push a menu for editing object classes.

edit_volume_multiplier(*ambiance: earwax.story.world.WorldAmbiance*) → Callable[[], Generator[None, None, None]]
Return a callable that can be used to set an ambiance volume multiplier.

Parameters **`ambiance`** – The ambiance whose volume multiplier will be changed.

`get_rooms`(*include_current: bool = True*) → List[earwax.story.world.WorldRoom]
Return a list of rooms from this world.

Parameters **`include_current`** – If this value is `True`, the current room will be included.

`goto_room`() → Generator[None, None, None]
Let the player choose a room to go to.

`object_actions`() → Generator[None, None, None]
Push a menu that lets you configure object actions.

`remessage`() → Optional[Generator[None, None, None]]
Set a message on the currently-focused object.

`rename`() → Generator[None, None, None]
Rename the currently focused object.

`reposition_object`() → None
Reposition the currently selected object.

`room`
Return the current room.

`save_world`() → None
Save the world.

`set_action_sound`(*action: earwax.story.world.WorldAction*) → Generator[None, None, None]
Set the sound on the given action.

Parameters **`action`** – The action whose sound will be changed.

`set_message`(*action: earwax.story.world.WorldAction*) → Generator[None, None, None]
Push an editor to set the message on the provided action.

Parameters **`action`** – The action whose message attribute will be modified.

`set_name`(*obj: Union[earwax.story.world.WorldAction, earwax.story.world.RoomObject, earwax.story.world.WorldRoom]*) → Generator[None, None, None]
Push an editor that can be used to change the name of `obj`.

Parameters **`obj`** – The object to rename.

`set_object_type`() → None
Change the type of an object.

`set_world_messages`() → Generator[None, None, None]
Push a menu that allows the editing of world messages.

`set_world_sound`(*name: str*) → Callable[[], Generator[None, None, None]]
Set the given sound.

Parameters **`name`** – The name of the sound to edit.

`shadow_description`() → None
Set the description of this room from another room.

`shadow_name`() → None
Sow a menu to select another room whose name will be shadowed.

`sounds_menu`() → Optional[Generator[None, None, None]]
Add or remove ambiances for the currently focused object.

`world_sounds`() → Generator[None, None, None]
Push a menu that can be used to configure world sounds.

class earwax.story.**ObjectPositionLevel**(*game: Game, object: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit], level: EditLevel, initial_position: Optional[earwax.story.world.DumpablePoint] = NOTHING*)

Bases: `earwax.level.Level`

A level for editing the position of an object.

Variables

- **object** – The object or exit whose position will be edited.
- **level** – The edit level which pushed this level.

backward() → None
Move the sound backwards.

cancel() → None
Undo the move, and return everything to how it was.

clear() → None
Clear the object position.

done() → None
Finish editing.

down() → None
Move the sound down.

forward() → None
Move the sound forwards.

get_initial_position() → Optional[earwax.story.world.DumpablePoint]
Get the object position.

left() → None
Move the sound left.

move(*x: int = 0, y: int = 0, z: int = 0*) → None
Change the position of this object.

reset() → None
Reset the current room.

right() → None
Move the sound right.

up() → None
Move the sound up.

class earwax.story.**PlayLevel**(*game: Game, world_context: StoryContext, cursor_sound: Optional[earwax.sound.Sound] = None, inventory: List[earwax.story.world.RoomObject] = NOTHING, reverb: Optional[GlobalFdnReverb] = None, object_ambiances: Dict[str, List[earwax.ambiance.Ambiance]] = NOTHING, object_tracks: Dict[str, List[earwax.track.Track]] = NOTHING*)

Bases: `earwax.level.Level`

A level that can be used to play a story.

Instances of this class can only play stories, not edit them.

Variables

- **world_context** – The context that contains the world, and the state for this story.
- **action_sounds** – The sounds which were started by object actions.
- **cursor_sound** – The sound that plays when moving through objects and ambiances.
- **inventory** – The list of `Roomobject` instances that the player is carrying.
- **reverb** – The reverb object for the current room.
- **object_ambiances** – The ambiances for a all objects in the room, excluding those in the players' `inventory`.
- **object_tracks** – The tracks for each object in the current room, excluding those objects that are in the player's `inventory`.

actions_menu(*obj: earwax.story.world.RoomObject*, *menu_action: Optional[earwax.story.world.WorldAction] = None*) → None

Show a menu of object actions.

Parameters

- **obj** – The object which the menu will be shown for.
- **menu_action** – The action which will be used instead of the default `actions_action`.

activate() → None

Activate the currently focussed object.

build_inventory() → None

Build the player inventory.

This method should be performed any time *state* changes.

cycle_category(*direction: int*) → Generator[None, None, None]

Cycle through information categories.

cycle_object(*direction: int*) → None

Cycle through objects.

do_action(*action: earwax.story.world.WorldAction*, *obj: Union[earwax.story.world.RoomObject, earwax.story.world.RoomExit]*, *pan: bool = True*) → None

Actually perform an action.

Parameters

- **action** – The action to perform.
- **obj** – The object that owns this action.

  If this value is of type *RoomObject*, and its `position` value is not `None`, then the action sound will be panned accordingly..
- **pan** – If this value evaluates to `False`, then regardless of the `obj` value, no panning will be performed.

drop_object(*obj: earwax.story.world.RoomObject*) → Callable[[], None]

Return a callable that can be used to drop an object.

drop_object_menu() → None

Push a menu that can be used to drop an object.

get_gain(*type: earwax.track.TrackTypes*, *multiplier: float*) → float

Return the proper gain.

get_objects() → List[earwax.story.world.RoomObject]
Return a list of objects that the player can see.

This method will exclude objects which are in the as yet unimplemented player inventory.

The resulting list will be sorted with Python's `sorted` builtin.

inventory_menu() → None
Show the inventory menu.

main_menu() → Generator[None, None, None]
Return to the main menu.

next_category() → Generator[None, None, None]
Next information category.

next_object() → None
Go to the next object.

object
Return the object from `self.state`.

object_menu(*obj: earwax.story.world.RoomObject*) → Callable[[], None]
Return a callable which shows the inventory menu for an object.

objects_menu(*objects: List[earwax.story.world.RoomObject], func: Callable[[earwax.story.world.RoomObject], Callable[[], None]], title: str*) → None
Push a menu of objects.

on_pop() → None
Stop all the action sounds.

on_push() → None
Set the initial room.

The room is the world from the *`state`* object, rather than the *`initial_room`*.

pause() → None
Pause All the currently-playing room sounds.

perform_action(*obj: earwax.story.world.RoomObject, action: earwax.story.world.WorldAction*) → Callable[[], None]
Return a function that will perform an object action.

This method is used by *`actions_menu()`* to allow the player to trigger object actions.

The inner method performs the following actions:

- Shows the action message to the player.
- **Plays the action sound. If `obj` has coordinates, the sound will be** heard at those coordinates.
- Pops the level to remove the actions menu from the stack.

Parameters

- **`obj`** – The object which has the action.
- **`action`** – The action which should be performed.

play_action_sound(*sound: str, position: Optional[earwax.point.Point] = None*) → None
Play an action sound.

Parameters

- **`sound`** – The filename of the sound to play.
- **`position`** – The position of the owning object.

  If this value is `None`, the sound will not be panned.

`play_cursor_sound`(*position: Optional[earwax.point.Point]*) → None
Play and set the cursor sound.

`play_object_ambiances`(*obj: earwax.story.world.RoomObject*) → None
Play all the ambiances for the given object.

Parameters **`obj`** – The object whose ambiances will be played.

`previous_category`() → Generator[None, None, None]
Previous information category.

`previous_object`() → None
Go to the previous object.

`save_state`() → None
Save the current state.

`set_room`(*room: earwax.story.world.WorldRoom*) → None
Move to a new room.

`state`
Return the current state.

`stop_action_sounds`() → None
Stop all action sounds.

`stop_object_ambiances`(*obj: earwax.story.world.RoomObject*) → None
Stop all the ambiances for the given object.

Parameters **`obj`** – The object whose ambiances will be stopped.

`take_object`(*obj: earwax.story.world.RoomObject*) → None
Take an object.

`use_exit`(*x: earwax.story.world.RoomExit*) → None
Use the given exit.

This method is called by the `activate()` method.

Parameters **`x`** – The exit to use.

`use_object`(*obj: earwax.story.world.RoomObject*) → Callable[[], None]
Return a callable that can be used to use an object.

`use_object_menu`() → None
Push a menu that allows using an object.

`world`
Get the attached world.

class `earwax.story.`**`StoryContext`**(*game: earwax.game.Game, world: earwax.story.world.StoryWorld, edit: bool = NOTHING, state: earwax.story.world.WorldState = NOTHING, errors: List[str] = NOTHING, warnings: List[str] = NOTHING*)

Bases: `object`

Holds references to various objects required to make a story work.

`before_run`() → None
Set the default panning strategy.

configure_earwax() → None
    Push a menu that can be used to configure Earwax.

configure_music() → None
    Allow adding and removing main menu music.

credit_menu(*credit: earwax.credit.Credit*) → Callable[[], None]
    Push a menu that can deal with credits.

credits_menu() → None
    Add or remove credits.

earwax_bug() → None
    Open the Earwax new issue URL.

get_default_config_file() → pathlib.Path
    Get the default configuration filename.

get_default_logger() → logging.Logger
    Return a default logger.

get_default_state() → earwax.story.world.WorldState
    Get a default state.

get_main_menu() → earwax.menus.menu.Menu
    Create a main menu for this world.

get_window_caption() → str
    Return a suitable window title.

load() → None
    Load an existing game, and start it.

play() → None
    Push the world level.

push_credits() → None
    Push the credits menu.

set_initial_room() → None
    Set the initial room.

set_panner_strategy() → None
    Allow the changing of the panner strategy.

show_warnings() → None
    Show any generated warnings.

world_options() → None
    Configure the world.

9.1.2 Submodules

earwax.action module

Provides the Action class.

class earwax.action.**Action**(*title: str, func: Callable[[], Optional[Generator[None, None, None]]], symbol: Optional[int] = None, mouse_button: Optional[int] = None, modifiers: int = 0, joystick_button: Optional[int] = None, hat_direction: Optional[Tuple[int, int]] = None, interval: Optional[float] = None*)

Bases: `object`

An action that can be called from within a game.

Actions can be added to `Level`, and `ActionMap` instances.

Usually, this class is not used directly, but returned by the `action()` method of whatever `Level` or `ActionMap` instance it is bound to.

Variables

- **`title`** – The title of this action.
- **`func`** – The function to run.

  If this value is a normal function, it will be called when the action is triggered.

  If this function is a generator, any code before the first `yield` statement will be run when the triggering key, hat, joystick button, or mouse button is pressed down. Anything after that will be run when the same trigger is released.

  It is worth noting that the behaviour of having a generator that yields more than once is undefined.
- **`symbol`** – The keyboard symbol to be used (should be one of the symbols from pyglet.window.key).
- **`mouse_button`** – The mouse button to be used (should be one of the symbols from pyglet.window.mouse).
- **`modifiers`** – Keyboard modifiers. Should be made up of modifiers from pyglet.window.key.
- **`joystick_button`** – The button that must be pressed on a game controller to trigger this action.

  The button can be any integer supported by any game pad.
- **`hat_direction`** – The position the hat must be in to trigger this action.

  This value must be a value supported by the hat control on the controller you're targetting.

  There are some helpful default values in *`earwax.hat_directions`*. If they do not suit your purposes, simply provide your own tuple.

  It is worth noting that if you rely on the hat, there are a few things to be aware of:

  If you rely on generators in hat-triggered actions, then all actions that have yielded will be stopped when the hat returns to its default position. This is because Earwax does not attempt to keep track of the last direction, and the hat does not generate release events like joystick buttons do.
- **`interval`** – How often this action can run.

  If `None`, then it is a one-time action. One-time actions should be used for things like quitting the game, or passing through exits, where multiple uses in a short space of time would be undesirable. Otherwise, it will be the number of seconds which must elapse between runs.
- **`last_run`** – The time this action was last run.

To get the number of seconds since an action was last run, use `time() - action.last_run`.

run (*dt: Optional[float]*) → Optional[Generator[None, None, None]]

Run this action.

This method may be called by `pyglet.clock.schedule_interval`.

If you need to know how an action has been called, you can override this method and check `dt`.

It will be `None` if it wasn't called by `schedule_interval`. This will happen either if you are dealing with a one-time action (`interval` is `None`), or the action is being called as soon as it is triggered (`schedule_interval` doesn't allow a function to be run and scheduled in one call).

If you need to call an action from your own code, you should use:

```
action.run(None)
```

Parameters **dt** – Refer to the documentation for pyglet.clock.

earwax.action_map module

Provides the ActionMap class.

class earwax.action_map.**ActionMap**

Bases: `object`

An object to hold actions.

This class is the answer to the question "What do I do when I have actions I want to be attached to multiple levels?"

You could of course use a for loop, but this class is quicker:

```
action_map: ActionMap = ActionMap()

@action_map.action(...)

@action_map.action(...)

level: Level = Level(game)
level.add_actions(action_map)
```

Variables **actions** – The actions to be stored on this map.

action (*title: str*, *symbol: Optional[int] = None*, *mouse_button: Optional[int] = None*, *modifiers: int = 0*, *joystick_button: Optional[int] = None*, *hat_direction: Optional[Tuple[int, int]] = None*, *interval: Optional[float] = None*) → Callable[[Callable[[], Optional[Generator[None, None, None]]]], earwax.action.Action]

Add an action to this object.

For example:

```
@action_map.action(
    'Walk forwards', symbol=key.W, mouse_button=mouse.RIGHT,
    interval=0.5
)
def walk_forwards():
    # ...
```

It is possible to use a generator function to have code executed before and after a trigger fires. If you need this behaviour, see the documentation for the `func` attribute of `earwax.Action`.

Parameters

- **`title`** – The `title` of the new action.

  This value is currently only used by `earwax.ActionMenu`.
- **`symbol`** – The resulting action's `symbol` attribute.
- **`mouse_button`** – The resulting action's `mouse_button` attribute.
- **`modifiers`** – The resulting action's `modifiers` attribute.
- **`joystick_button`** – The resulting action's `joystick_button` attribute.
- **`hat_direction`** – The resulting action's `hat_direction` attribute.
- **`interval`** – The resulting action's `interval` attribute.

`add_actions`(*action_map: earwax.action_map.ActionMap*) → None

Add the actions from the provided map to this map.

Parameters **`action_map`** – The map whose actions should be appended to this one.

earwax.ambiance module

Provides the Ambiance class.

`class` `earwax.ambiance.`**`Ambiance`**(*protocol: str*, *path: str*, *coordinates: earwax.point.Point*)

Bases: `object`

A class that represents a positioned sound on a map.

If you want to know more about the `stream` and `path` attributes, see the documentation for `synthizer.StreamingGenerator`.

Variables

- **`protocol`** – The `protocol` argument to pass to `synthizer.StreamingGenerator``.`
- **`path`** – The `path` argument to pass to `synthizer.StreamingGenerator`.
- **`coordinates`** – The coordinates of this ambiance.
- **`sound`** – The playing sound.

  This value is initialised as part of the `play()` method.

`classmethod from_path`(*path: pathlib.Path*, *coordinates: earwax.point.Point*) → earwax.ambiance.Ambiance

Return a new instance from a path.

Parameters

- **`path`** – The path to build the ambiance from.

  If this value is a directory, then a random file will be chosen.
- **`coordinates`** – The coordinates of this ambiance.

`play`(*sound_manager: earwax.sound.SoundManager*, ***kwargs*) → None

Load and position the sound.

Parameters

- **sound_manager** – The sound manager which will be used to play this ambiance.
- **kwargs** – The additional keyword arguments to pass to `play_path()`.

stop() → None

Stop this ambiance from playing.

earwax.config module

Provides the Config and ConfigValue classes.

class earwax.config.**Config**

Bases: `object`

Holds configuration subsections and values.

Any attribute that is an instance of `earwax.Config` is considered a subsection.

Any attribute that is an instance of `earwax.ConfigValue` is considered a configuration value.

You can create sections like so:

```
from earwax import Config, ConfigValue

class GameConfig(Config):
    '''Example configuration page.'''

    hostname = ConfigValue('localhost')
    port = ConfigValue(1234)

c = GameConfig()
```

Then you can access configuration values like this:

```
host_string = f'{c.hostname.value}:{c.port.value}'
# ...
```

Use the `dump()` method to get a dictionary suitable for dumping with json.

To set the name that will be used by `earwax.ConfigMenu`, subclass `earwax.Config`, and include a *__section_name__* attribute:

```
class NamedConfig(Config):
    __section_name__ = 'Options'
```

Variables **__section_name__** – The human-readable name of this section.

At present, this attribute is only used by `earwax.ConfigMenu`.

dump() → Dict[str, Any]

Return all configuration values, recursing through subsections.

For example:

```
c = ImaginaryConfiguration()
d = c.dump()
with open('config.yaml', 'w') as f:
    json.dump(d, f)
```

Use the `populate_from_dict()` method to restore dumped values.

load (*f: TextIO*) → None

Load data from a file.

Uses the `populate_from_dict()` method on dataloaded from the given file:

```
c = ImaginaryConfigSection()
with open('config.yaml', 'r'):
    c.load(f)
```

To save the data in the first place, use the `save()` method.

Parameters **f** – A file-like object to load data from.

populate_from_dict (*data: Dict[str, Any]*) → None

Populate values from a dictionary.

This function is compatible with (and used by) `dump()`:

```
c = Config()
with open('config.yaml', 'r') as f:
    c.populate_from_dict(json.load(f))
```

Any missing values from *data* are ignored.

Parameters **data** – The data to load.

save (*f: TextIO*) → None

Dump this configuration section to a file.

Uses the `dump()` method to get the dumpable data.

You can save a configuration section like so:

```
c = ImaginaryConfigSection()
with open('config.yaml', 'w') as f:
    c.save(f)
```

By default, YAML is used.

Parameters **f** – A file-like object to write the resulting data to.

class earwax.config.**ConfigValue** (*value: T, name: Optional[str] = None, type_: Optional[object] = None, value_converters: Optional[Dict[object, Callable[[ConfigValue], str]]] = None, dump_func: Optional[Callable[[T], T]] = None, load_func: Optional[Callable[[str], T]] = None*)

Bases: `typing.Generic`

A configuration value.

This class is used to make configuration values:

```
name = ConfigValue('username', name='Your character name', type_=str)
```

If you are dealing with a non-standard object, you can set custom functions for loading and dumping the objects:

```
from pathlib import Path
option = ConfigValue(Path.cwd(), name='Some directory')

@option.dump
def dump_path(value: Path) -> str:
    return str(value)
```

(continues on next page)

(continued from previous page)

```
@option.load
def load_path(value: str) -> Path:
    return Path(value)
```

Variables

- **`value`** – The value held by this configuration value.
- **`name`** – The human-readable name of this configuration value.

  The name is currently only used by `earwax.ConfigMenu`.
- **`type_`** – The type of this value. Can be inferred from `value`.

  Currently this attribute is used by `earwax.ConfigMenu` to figure out how to construct the widget that will represent this value.
- **`value_converters`** – A dictionary of `type`: `converter` functions.

  These are used by `earwax.ConfigMenu.option_menu()` to print `value`, instead of `value_to_string()`.
- **`default`** – The default value for this configuration value.

  This will be inferred from `value`.
- **`dump_func`** – A function that will take the actual value, and return something that YAML can dump.
- **`load_func`** – A function that takes the value that was loaded by YAML, and returns the actual value.

`dump`(*func: Callable[[T], T]*) → Callable[[T], T]
Add a dump function.

Parameters **`func`** – The function that will be decorated.

See the description for `dump_func`.

`load`(*func: Callable[[str], T]*) → Callable[[str], T]
Add a load function.

Parameters **`func`** – The function that will be decorated.

See the description for `load_func`.

`value_to_string`() → str
Return `value` as a string.

This method is used by `earwax.ConfigMenu` when it shows values.

earwax.configuration module

Provides the Config class.

class `earwax.configuration.`**`EarwaxConfig`**
Bases: *`earwax.config.Config`*

The main earwax configuration.

An instance of this value will be loaded to `earwax.Game.config`.

It is advised to configure the game before calling `earwax.Game.run()`.

`editors = <earwax.configuration.EditorConfig object>`

`menus = <earwax.configuration.MenuConfig object>`

`sound = <earwax.configuration.SoundConfig object>`

`speech = <earwax.configuration.SpeechConfig object>`

class `earwax.configuration.`**`EditorConfig`**

Bases: *`earwax.config.Config`*

Configure various things about editors.

Variables **`hat_alphabet`** – The letters that can be entered by a controller's hat.

`hat_alphabet = ConfigValue(value=' abcdefghijklmnopqrstuvwxyz.,1234567890@ABCDEFGHIJKL`

class `earwax.configuration.`**`MenuConfig`**

Bases: *`earwax.config.Config`*

The menu configuration section.

Variables

- **`default_item_select_sound`** – The default sound to play when a menu item is selected.

  If this value is `None`, no sound will be played, unless specified by the selected menu item.

- **`default_item_activate_sound`** – The default sound to play when a menu item is activated.

  If this value is `None`, no sound will be played, unless specified by the activated menu item.

`default_item_activate_sound = ConfigValue(value=None, name='The default sound that pla`

`default_item_select_sound = ConfigValue(value=None, name='The default sound that plays`

class `earwax.configuration.`**`SoundConfig`**

Bases: *`earwax.config.Config`*

Configure various aspects of the sound system.

Variables

- **`master_volume`** – The volume of `audio_context`.

  This value acts as a master volume, and should be changed with either `adjust_volume()`, or `set_volume()`.

- **`max_volume`** – The maximum volume allowed by `adjust_volume()`.

- **`sound_volume`** – The volume of general sounds.

  This volume is used by earwax to set the volume of `interface_sound_manager` values.

- **`music_volume`** – The volume of game music.

  Earwax uses this value to set the volume of the `music_sound_manager` sound manager.

- **`ambiance_volume`** – The volume of game ambiances.

  Earwax uses this value to set the volume of the `ambiance_sound_manager` sound manager.

- **`default_cache_size`** – The default size (in bytes) for the default `buffer_cache` object.

`ambiance_volume = ConfigValue(value=0.4, name='Ambiance volume', type_=<class 'float'>`

`default_cache_size = ConfigValue(value=524288000, name='The size of the default sound`

`master_volume = ConfigValue(value=1.0, name='Master volume', type_=<class 'float'>, va`

`max_volume = ConfigValue(value=1.0, name='Maximum volume', type_=<class 'float'>, valu`

`music_volume = ConfigValue(value=0.4, name='Music volume', type_=<class 'float'>, valu`

`sound_volume = ConfigValue(value=0.5, name='Sound volume', type_=<class 'float'>, valu`

`class` `earwax.configuration.`**`SpeechConfig`**

Bases: *`earwax.config.Config`*

The speech configuration section.

Variables

- **`speak`** – Whether or not calls to `output()` will produce speech.
- **`braille`** – Whether or not calls to `output()` will produce braille.

`braille = ConfigValue(value=True, name='Braille', type_=<class 'bool'>, value_converte`

`speak = ConfigValue(value=True, name='Speech', type_=<class 'bool'>, value_converters=`

`earwax.configuration.`**`dump_path`**(*value: Optional[pathlib.Path]*) → Optional[str]

Return a path as a string.

Parameters **`value`** – The path to convert.

`earwax.configuration.`**`load_path`**(*value: Optional[str]*) → Optional[pathlib.Path]

Load a path from a string.

Parameters **`value`** – The string to convert to a path.

earwax.credit module

Provides the Credit class.

`class` `earwax.credit.`**`Credit`**(*name: str*, *url: str*, *sound: Optional[pathlib.Path] = None*, *loop: bool = True*)

Bases: `object`

A credit in a game.

Variables

- **`name`** – The name of the person or company who is being credited.

  This value will be shown in a menu generated by `earwax.Menu.from_credits()`.
- **`url`** – The URL to open when this credit is selected.
- **`sound`** – An optional sound to play while this credit is shown.
- **`loop`** – Whether ot not to loop `sound`.

`classmethod earwax_credit``()` → earwax.credit.Credit

Get an earwax credit.

earwax.dialogue_tree module

Provides the DialogueLine and DialogueTree classes.

class `earwax.dialogue_tree.`**`DialogueLine`**(*parent: DialogueTree*, *text: Optional[str] = None*, *sound: Optional[pathlib.Path] = None*, *can_show: Optional[Callable[[], bool]] = None*, *on_activate: Optional[Callable[[], bool]] = None*, *responses: List[DialogueLine] = NOTHING*)

Bases: `object`

A line of dialogue.

Parameters

- **`parent`** – The dialogue tree that this line of dialogue belongs to.
- **`text`** – The text that is shown as part of this dialogue line.
- **`sound`** – A portion of recorded dialogue.
- **`can_show`** – A callable which will determine whether or not this line is visible in the conversation.

  If it returns `True`, this line will be shown in the list.
- **`on_activate`** – A callable which will be called when this line is selected from the list of lines.

  If it returns `True`, the conversation can continue.
- **`responses`** – A list of responses to this line.

class `earwax.dialogue_tree.`**`DialogueTree`**(*tracks: List[earwax.track.Track] = NOTHING*)

Bases: `object`

A dialogue tree object.

Variables

- **`children`** – The top-level dialogue lines for this instance.
- **`tracks`** – A list of tracks to play while this dialogue tree is in focus.

`get_children`() → List[earwax.dialogue_tree.DialogueLine]

Get a list of all the children who can be shown currently.

This method returns a list of those children for whom `child.can_show()` is `True`.

earwax.die module

Provides the Die class.

class `earwax.die.`**`Die`**(*sides: int = 6*)

Bases: *`earwax.mixins.RegisterEventMixin`*

A single dice.

Variables **`sides`** – The number of sides this die has.

`on_roll`(*value: int*) → None

Code to be run when a die is rolled.

An event which is dispatched by `roll()` method.

Parameters **`value`** – The number that has been rolled.

`roll``()` → int
Roll a die.

Returns a number between 1, and `self.size`.

earwax.editor module

Provides the Editor class.

class `earwax.editor.`**`Editor`**(*game: Game*, *dismissible: bool = True*, *text: str = ”*, *cursor_position: Optional[int] = None*, *vertical_position: Optional[int] = None*)
Bases: *`earwax.level.Level`*, *`earwax.mixins.DismissibleMixin`*

A basic text editor.

By default, the enter key dispatches the `on_submit` event, with the contents of `earwax.Editor.text`.

Below is an example of how to use this class:

```
e: Editor = Editor(game)

@e.event
def on_submit(text: str) -> None:
    # Do something with text...

game.push_level(e)
```

Variables

- **`func`** – The function which should be called when pressing enter in an edit field.
- **`text`** – The text which can be edited by this object.
- **`cursor_position`** – The position of the cursor.
- **`vertical_position`** – The position in the alphabet of the hat.

`beginning_of_line``()` → None
Move to the start of the current line.

By default, this method is called when the home key is pressed.

`clear``()` → None
Clear this editor.

By default, this method is called when control + u is pressed.

`copy``()` → None
Copy the contents of this editor to the clipboard.

`cut``()` → None
Cut the contents of this editor to the clipboard.

`do_delete``()` → None
Perform a forward delete.

Used by `motion_delete()`, as well as the vertical hat movement methods.

`echo`(*text: str*) → None
Speak the provided text.

Parameters **`text`** – The text to speak, using `tts.speak`.

`echo_current_character``()` → None
Echo the current character.

Used when moving through the text.

`end_of_line``()` → None
Move to the end of the line.

By default, this method is called when the end key is pressed.

`hat_down``()` → None
Move down through the list of letters.

`hat_up``()` → None
Change the current letter to the previous one in the configured alphabet.

If the cursor is at the end of the line, moving up will select a “save” button.

If the cursor is not at the end of the line, moving up will select a “delete” button.

`insert_text``(`*text: str*`)` → None
Insert `text` at the current cursor position.

`motion_backspace``()` → None
Delete the previous character.

This will do nothing if the cursor is at the beginning of the line, or there is no text to delete.

`motion_delete``()` → None
Delete the character under the cursor.

Nothing will happen if we are at the end of the line (or there is no text, which will amount to the same thing).

`motion_down``()` → None
Arrow down.

Since we’re not bothering with multiline text fields at this stage, just move the cursor to the end of the line, and read the whole thing.

By default, this method is called when the down arrow key is pressed.

`motion_left``()` → None
Move left in the editor.

By default, this method is called when the left arrow key is pressed.

`motion_right``()` → None
Move right in the editor.

By default, this method is called when the right arrow key is pressed.

`motion_up``()` → None
Arrow up.

Since we’re not bothering with multiline text fields at this stage, just move the cursor to the start of the line, and read the whole thing.

By default, this method is called when the up arrow key is pressed.

`on_submit``(`*text: str*`)` → None
Code to be run when this editor is submitted.

The event which is dispatched if the enter key is pressed.

Parameters **`text`** – The contents of `self.text`.

`on_text`(*text: str*) → None
Text has been entered.

If the cursor is at the end of the line, append the text. Otherwise, insert it.

Parameters **`text`** – The text that has been entered.

`paste`() → None
Paste the contents of the clipboard into this editor.

`set_cursor_position`(*pos: Optional[int]*) → None
Set the cursor position within `text`.

If `pos` is `None`, then the cursor will be at the end of the line. Otherwise, `pos` should be an integer between 0 and `len(self.text) - 1`.

Parameters **`pos`** – The new cursor position.

`submit`() → None
Submit `self.text`.

Dispatch the `on_submit` event with the contents of `self.text`.

By default, this method is called when the enter key is pressed.

earwax.event_matcher module

Provides the EventMatcher class.

`class` `earwax.event_matcher.`**`EventMatcher`**(*game: Game*, *name: str*)
Bases: `object`

Matches events for `Game` instances.

An object to call events on a `Game` instance's `level` property.

Used to prevent us writing loads of events out.

Variables

- **`game`** – The game this matcher is bound to.
- **`name`** – The name of the event this matcher uses.

`dispatch`(**args*, ***kwargs*) → None
Dispatch this event.

Find the appropriate event on game.level, if game.level is not None.

If `self.game.level` doesn't have an event of the proper name, search instead on `self.game`.

Parameters

- **`args`** – The positional arguments to pass to any event that is found.
- **`kwargs`** – The keyword arguments to pass to any event that is found.

earwax.game module

Provides the Game class.

class earwax.game.**Game**(*name: str = 'earwax.game', audio_context: Optional[object] = NOTHING, buffer_cache: earwax.sound.BufferCache = NOTHING, interface_sound_manager: earwax.sound.SoundManager = NOTHING, music_sound_manager: Optional[earwax.sound.SoundManager] = NOTHING, ambiance_sound_manager: Optional[earwax.sound.SoundManager] = NOTHING, thread_pool: concurrent.futures._base.Executor = NOTHING, credits: List[earwax.credit.Credit] = NOTHING, logger: logging.Logger = NOTHING*)

Bases: `earwax.mixins.RegisterEventMixin`

The main game object.

This object holds a reference to the game window, as well as a list of Level instances.

In addition, references to various parts of the audio subsystem reside on this object, namely `audio_context`.

Instances of the Level class can be pushed, popped, and replaced. The entire stack can also be cleared.

Although it doesn't matter in what order you create objects, a `Game` instance is necessary for `Level` instances - and subclasses thereof - to be useful.

Variables

- **window** – The pyglet window used to display the game.
- **config** – The configuration object used by this game.
- **name** – The name of this game. Used by `get_settings_path()`.
- **audio_context** – The Synthizer context to route audio through.
- **interface_sound_manager** – A sound manager for playing interface sounds.
- **music_sound_manager** – A sound manager for playing music.
- **ambiance_sound_manager** – A sound manager for playing ambiances.
- **levels** – All the pushed `earwax.Level` instances.
- **triggered_actions** – The currently triggered `earwax.Action` instances.
- **key_release_generators** – The `earwax.Action` instances which returned generators, and need to do something on key release.
- **mouse_release_generators** – The `earwax.Action` instances which returned generators, and need to do something on mouse release.
- **joybutton_release_generators** – The `earwax.Action` instances which returned generators, and need to do something on joystick button release.
- **event_matchers** – The `earwax.EventMatcher` instances used by this object.

  To take advantage of the pyglet events system, subclass `earwax.Game`, or `earwax.Level`, and include events from pyglet.window.Window.
- **joysticks** – The list of joysticks that have been opened by this instance.
- **thread_pool** – An instance of `ThreadPoolExecutor` to use for threaded operations.
- **tasks** – A list of `earwax.Task` instances.

  You can add tasks with the `register_task()` decorator, and remove them again with the `remove_task()` method.

adjust_volume(*amount: float*) → float

Adjust the master volume.

Parameters **`amount`** – The amount to add to the current volume.

`after_run``()` → None

Run code before the game exits.

This event is dispatched after the main game loop has ended.

By this point, synthizer has been shutdown, and there is nothing else to be done.

`before_run``()` → None

Do stuff before starting the main event loop.

This event is used by the run method, before any initial level is pushed, or any of the sound managers are created.

This is the event to use if you're planning to load configuration.

By this point, default events have been decorated, such as on_key_press and on_text. Also, we are inside a synthizer.initialized context manager, so feel free to play sounds, and use `self.audio_context`.

`cancel``(`*message: str = 'Cancelled', level: Optional[earwax.level.Level] = None*`)` → None

Cancel with an optional message.

All this method does is output the given message, and either pop the most recent level, or reveal the given level.

Parameters

- **`message`** – The message to output.
- **`level`** – The level to reveal.

  If this value is `None`, then the most recent level will be popped.

`change_volume``(`*amount: float*`)` → Callable[[], None]

Return a callable that can be used to change the master volume.

Parameters **`amount`** – The amount to change the volume by.

`clear_levels``()` → None

Pop all levels.

The `earwax.Level.on_pop()` method will be called on every level that is popped.

`click_mouse``(`*button: int, modifiers: int*`)` → None

Simulate a mouse click.

This method is used for testing, to simulate first pressing, then releasing a mouse button.

Parameters

- **`button`** – One of the mouse button constants from pyglet.window.mouse.
- **`modifiers`** – One of the modifier constants from pyglet.window.key.

`finalise_run``()` → None

Perform the final steps of running the game.

- Dispatch the `before_run()` event.
- Call `pyglet.app.run()`.
- Unload Cytolk.
- Dispatch the `after_run()` event.

`get_default_buffer_cache``()` → earwax.sound.BufferCache

Return the default buffer cache.

Parameters **`instance`** – The game to return the buffer cache for.

`get_default_logger``()` → logging.Logger
Return a logger.

`get_settings_path``()` → pathlib.Path
Get a path to store game settings.

Uses `pyglet.resource.get_settings_path` to get an appropriate settings path for this game.

`init_sdl``()` → None
Initialise SDL.

`level`
Get the most recently added `earwax.Level` instance.

If the stack is empty, `None` will be returned.

`on_close``()` → None
Run code when closing the window.

Called when the window is closing.

This is the default event that is used by `pyglet.window.Window`.

By default, this method calls `self.clear_levels()`, to ensure any clean up code is called.

`on_joybutton_press`(*joystick: object*, *button: int*) → bool
Handle the press of a joystick button.

This is the default handler that fires when a joystick button is pressed.

Parameters **`joystick`** – The joystick that emitted the event.

: param button: The button that was pressed.

`on_joybutton_release`(*joystick: object*, *button: int*) → bool
Handle the release of a joystick button.

This is the default handler that fires when a joystick button is released.

Parameters **`joystick`** – The joystick that emitted the event.

: param button: The button that was pressed.

`on_joyhat_motion`(*joystick: object*, *x: int*, *y: int*) → bool
Handle joyhat motions.

This is the default handler that fires when a hat is moved.

If the given position is the default position `(0, 0)`, then any actions started by hat motions are stopped.

Parameters **`joystick`** – The joystick that emitted the event.

: param x: The left / right position of the hat.

: param y: The up / down position of the hat.

`on_key_press`(*symbol: int*, *modifiers: int*) → bool
Handle a pressed key.

This is the default event that is used by `pyglet.window.Window`.

By default it iterates through `self.level.actions`, and searches for events that match the given symbol and modifiers.

Parameters

- **symbol** – One of the key constants from pyglet.window.key.
- **modifiers** – One of the modifier constants from pyglet.window.key.

on_key_release (*symbol: int*, *modifiers: int*) → bool

Handle a released key.

This is the default event that is used by `pyglet.window.Window`.

Parameters

- **symbol** – One of the key constants from pyglet.window.key.
- **modifiers** – One of the modifier constants from pyglet.window.key.

on_mouse_press (*x: int*, *y: int*, *button: int*, *modifiers: int*) → bool

Handle a mouse button press.

This is the default event that is used by `pyglet.window.Window`.

By default, this method pretty much acts the same as `on_key_press()`, except it checks the discovered actions for mouse buttons, rather than symbols.

Parameters

- **x** – The x coordinate of the mouse.
- **y** – The y coordinate of the mouse.
- **button** – One of the mouse button constants from pyglet.window.mouse.
- **modifiers** – One of the modifier constants from pyglet.window.key.

on_mouse_release (*x: int*, *y: int*, *button: int*, *modifiers: int*) → bool

Handle a mouse button release.

This is the default event that is used by `pyglet.window.Window`.

By default, this method is pretty much the same as `on_key_release()`, except that it uses the discovered actions mouse button information.

Parameters

- **x** – The x coordinate of the mouse.
- **y** – The y coordinate of the mouse.
- **button** – One of the mouse button constants from pyglet.window.mouse.
- **modifiers** – One of the modifier constants from pyglet.window.key.

open_joysticks () → None

Open and attach events to all attached joysticks.

output (*text: str*, *interrupt: bool = False*) → None

Output braille and / or speech.

The earwax configuration is used to determine what should be outputted.

Parameters

- **text** – The text to be spoken or output to a braille display.
- **interrupt** – If Whether or not to silence speech before outputting anything else.

poll_synthizer_events (*dt: float*) → None

Poll the audio context for new synthizer events.

Parameters **dt** – The delta provided by Pyglet.

pop_level() → None

Pop the most recent `earwax.Level` instance from the stack.

If there is a level underneath the current one, then events will be passed to it. Otherwise there will be an empty stack, and events won't get handled.

This method calls `on_pop()` on the popped level, and `on_reveal()` on the one below it.

pop_levels(*n: int*) → None

Pop the given number of levels.

Parameters **n** – The number of times to call `pop_level()`.

press_key(*symbol: int*, *modifiers: int*, *string: Optional[str] = None*, *motion: Optional[int] = None*) → None

Simulate a key press.

This method is used in tests.

First presses the given key combination, then releases it.

If string and motion are not None, then on_text, and on_text_motion events will also be fired.

Parameters

- **symbol** – One of the key constants from pyglet.window.key.
- **modifiers** – One of the modifier constants from pyglet.window.key.
- **string** – A string to be picked up by an `on_text` event handler..
- **motion** – A key to be picked up by an `on_text_motion` event handler.

push_action_menu(*title: str = 'Actions'*, ***kwargs*) → earwax.menus.action_menu.ActionMenu

Push and return an action menu.

This method reduces the amount of code required to create a help menu:

```
@level.action(
    'Help Menu', symbol=key.SLASH, modifiers=key.MOD_SHIFT
)
def help_menu() -> None:
    game.push_action_menu()
```

Parameters

- **title** – The title of the new menu.
- **kwargs** – The extra keyword arguments to pass to the ActionMenu constructor.

push_credits_menu(*title='Game Credits'*) → earwax.menus.menu.Menu

Push a credits menu onto the stack.

This method reduces the amount of code needed to push a credits menu:

```
@level.action('Show credits', symbol=key.F1)
def show_credits() -> None:
    game.push_credits_menu()
```

Parameters **title** – The title of the new menu.

push_level(*level: earwax.level.Level*) → None
Push a level onto `self.levels`.

This ensures that all events will be handled by the provided level until another level is pushed on top, or the current one is popped.

This method also dispatches the `on_push()` event on the provided level.

If the old level is not None, then the `on_cover` event is dispatched on the old level, with the new level as the only argument.

Parameters **level** – The `earwax.Level` instance to push onto the stack.

register_task(*interval: Callable[[], float]*) → Callable[[Callable[[float], None]], earwax.task.Task]
Decorate a function to use as a task.

This function allows you to convert a function into a `Task` instance, so you can add tasks by decoration:

```
@game.register_task(lambda: uniform(1.0, 5.0))
def task(dt: float) -> None:
    '''A task.'''
    print('Working: %.2f.' % dt)
task.start()
```

Parameters **interval** – The function to use for the interval.

remove_task(*task: earwax.task.Task*) → None
Stop and remove a task.

Parameters **task** – The task to be stopped.

The task will first have its `stop()` method called, then it will be removed from the `tasks` list.

replace_level(*level: earwax.level.Level*) → None
Pop the current level, then push the new one.

This method uses `pop_level()`, and `push_level()`, so make sure you familiarise yourself with what events will be called on each level.

Parameters **level** – The `earwax.Level` instance to push onto the stack.

reveal_level(*Level: earwax.level.Level*) → int
Pop levels until `level` is revealed.

This method returned the number of levels which were popped.

Parameters **level** – The level to reveal.

run(*window: object*, *mouse_exclusive: bool = True*, *initial_level: Optional[earwax.level.Level] = None*) → None
Run the game.

By default, this method will perform the following actions in order:

- **Iterate over all the found event types on `pyglet.window.Window`,** and decorate them with `EventMatcher` instances. This means `Game` and `Level` subclasses can take full advantage of all event types by simply adding methods with the correct names to their classes.
- Load cytolk.
- Initialise SDL2.

- Set the requested mouse exclusive mode on the provided window.
- call `open_joysticks()`.
- **If no `audio_context` is present, enter a** `synthizer.initialized` contextmanager.
- Call the `setup_run()` method.
- Call the `finalise_run()` method.

Parameters

- **`window`** – The pyglet window that will form the game's interface.
- **`mouse_exclusive`** – The mouse exclusive setting for the window.
- **`initial_level`** – A level to push onto the stack.

`set_volume`(*value: float*) → None
Set the master volume to a specific value.

Parameters **`value`** – The new volume.

`setup`() → None
Set up things needed for the game.

This event is dispatched just inside the synthizer context manager, before the various sound managers have been created.

This event is perfect for loading configurations ETC.

`setup_run`(*initial_level: Optional[earwax.level.Level]*) → None
Get ready to run the game.

This method dispatches the `setup()` event, and sets up sound managers.

Finally, it pushes the initial level, if necessary.

Parameters **`initial_level`** – The initial level to be pushed.

`start_action`(*a: earwax.action.Action*) → Optional[Generator[None, None, None]]
Start an action.

If the action has no interval, it will be ran straight away. Otherwise, it will be added to `self.triggered_actions`, and only ran if enough time has elapsed since the last run.

This method is used when a trigger fires - such as a mouse button or key sequence being pressed - that triggers an action.

Parameters **`a`** – The `earwax.Action` instance that should be started.

`start_rumble`(*joystick: object*, *value: float*, *duration: int*) → None
Start a simple rumble.

Parameters

- **`joystick`** – The joystick to rumble.
- **`value`** – A value from 0.0 to 1.0, which is the power of the rumble.
- **`duration`** – The duration of the rumble in milliseconds.

`stop`() → None
Close `self.window`.

If `self.window` is `None`, then :class:earwax.GameNotRunning' will be raised.

stop_action(*a: earwax.action.Action*) → None
Unschedule an action.

The provided action will be removed from `triggered_actions`.

This method is called when the user stops doing something that previously triggered an action, such as releasing a key or a mouse button

Parameters **a** – The `earwax.Action` instance that should be stopped.

stop_rumble(*joystick: object*) → None
Cancel a rumble.

Parameters **joystick** – The joystick you want to rumble.

exception earwax.game.**GameNotRunning**
Bases: `Exception`

This game is not running.

earwax.game_board module

Provides the GameBoard class.

class earwax.game_board.**GameBoard**(*game: Game, size: earwax.point.Point[int][int], tile_builder: Callable[[earwax.point.Point], T], coordinates: earwax.point.Point[int][int] = NOTHING*)
Bases: `earwax.level.Level`, `typing.Generic`

A useful starting point for making board games.

Tiles can be populated with the `populate()` method. This method will be called as part of the default `on_push()` event.

Variables

- **size** – The size of this board.

  This value will be the maximum possible coordinates on the board, with `(0, 0, 0)` being the minimum.

- **tile_builder** – The function that is used to build the GameBoard.

  The return value of this function should be of type `T`.

- **coordinates** – The coordinates of the player on this board.
- **tiles** – All the tiles generated by `populate()`.
- **populated_points** – All the points that have been populated by `populate()`.

current_tile
Return the current tile.

Gets the tile at the current coordinates.

If no such tile is found, `None` is returned.

get_tile(*p: earwax.point.Point[int][int]*) → T
Return the tile at the given point.

If there is no tile found, then `NoSuchTile` is raised.

Parameters **p** – The coordinates of the desired tile.

move(*direction: earwax.point.PointDirections*, *wrap: bool = False*) → Callable[[], None]
Return a callable that can be used to move the player.

For example:

```
board = GameBoard(...)

board.action(
    'Move left', symbol=key.LEFT
)(board.move(PointDirections.west))
```

Parameters

- **direction** – The direction that this action should move the player in.
- **wrap** – If `True`, then coordinates that are out of range will result in wrapping around to the other side of the board..

on_move_fail(*direction: earwax.point.PointDirections*) → None
Run code when the player fails to move.

An event that is dispatched when a player fails to move in the given direction.

Parameters **direction** – The direction the player tried to move in.

on_move_success(*direction: earwax.point.PointDirections*) → None
Handle a successful move.

An event that is dispatched by `move()`.

Parameters **direction** – The direction the player just moved.

on_push() → None
Populate the board.

populate() → None
Fill the board.

exception earwax.game_board.**NoSuchTile**
Bases: `Exception`

No such tile exists.

This exception is raised by `earwax.GameBoard.get_tile()` when no tile is found at the given coordinates.

earwax.hat_directions module

Provides hat motions to be used as shortcuts.

earwax.input_modes module

Provides the InputModes enumeration.

class earwax.input_modes.**InputModes**
Bases: `enum.Enum`

The possible input modes.

This enumeration is used to show appropriate triggers in `earwax.ActionMenu` instances.

Variables

- **`keyboard`** – The user is entering commands via keyboard or mouse.
- **`controller`** – The user is using a games controller.

`controller = 1`

`keyboard = 0`

earwax.level module

Provides classes for working with levels.

class `earwax.level.`**`IntroLevel`**(*game: Game*, *level: earwax.level.Level*, *sound_path: pathlib.Path*, *skip_after: Optional[float] = None*, *looping: bool = False*, *sound_manager: Optional[earwax.sound.SoundManager] = NOTHING*, *play_kwargs: Dict[str, Any] = NOTHING*)

Bases: `earwax.level.Level`

An introduction level.

This class represents a level that plays some audio, before optionally replacing itself in the level stack with `self.level`.

If you want it to be possible to skip this level, add a trigger for the `skip()` action.

Variables

- **`level`** – The level that will replace this one.
- **`sound_path`** – The sound to play when this level is pushed.
- **`skip_after`** – An optional number of seconds to wait before skipping this level.

  If this value is `None`, then the level will not automatically skip itself, and you will have to provide some other means of getting past it.
- **`looping`** – Whether or not the playing sound should loop.

  If this value is `True`, then `skip_after` must be `None`, otherwise `AssertionError` will be raised.
- **`sound_manager`** – The sound manager to use to play the sound.

  If this value is `None`, then the sound will not be playing.

  This value default to `earwax.Game.interface_sound_manager`.
- **`play_kwargs`** – Extra arguments to pass to the `play()` method of the `sound_manager`.

  When the `on_push()` event is dispatched, an error will be raised if this dictionary contains a `looping` key, as 2 `looping` arguments would be passed to `self.sound_manager.play_path`.
- **`sound`** – The sound object which represents the playing sound.

  If this value is `None`, then the sound will not be playing.

`get_default_sound_manager`() → Optional[earwax.sound.SoundManager]

Return a suitable sound manager.

`on_pop`() → None

Destroy any created `sound()`.

on_push() → None
Run code when this level has been pushed.

Starts playing `self.sound_path`, and optionally schedules an automatic skip.

skip() → Generator[None, None, None]
Skip this level.

Replaces this level in the level stack with `self.level`.

class earwax.level.**Level**(*game: Game*)
Bases: *earwax.mixins.RegisterEventMixin*, *earwax.action_map.ActionMap*

A level in a `Game` instance.

An object that contains event handlers. Can be pushed and pulled from within a `Game` instance.

While the `Game` object is the centre of a game, *Level* instances are where the magic happens.

If the included `action()` and `motion()` decorators aren't enough for your needs, and you want to harness the full power of the Pyglet event system, simply subclass `earwax.Level`, and include the requisite events. The underlying `Game` object will do all the heavy lifting for you, by way of the `EventMatcher` framework.

Variables

- **game** – The game this level is bound to.
- **actions** – A list of actions which can be called on this object. To define more, use the `action()` decorator.
- **motions** – The defined motion events. To define more, use the `motion()` decorator.
- **ambiances** – The ambiances for this level.
- **tracks** – The tracks (musical or otherwise) that play while this level is top of the stack.

motion(*motion: int*) → Callable[[MotionFunctionType], MotionFunctionType]
Add a handler to `motions`.

For example:

```
@level.motion(key.MOTION_LEFT)
def move_left():
    # ...
```

This is the method used by `earwax.Editor`, to make text editable, and `earwax.Menu`, to make menus searchable.

Parameters motion – One of the motion constants from pyglet.window.key.

on_cover(*level: earwax.level.Level*) → None
Code to run when this level has been covered by a new one.

on_pop() → None
Run code when this level is popped.

This event is called when a level has been popped from the `level stack` of a game.

on_push() → None
Run code when this level is pushed.

This event is called when a level has been pushed onto the `level stack` of a game.

on_reveal() → None
Code to be run when this level is exposed.

This event is called when the level above this one in the stack has been popped, thus revealing this level.

on_text_motion(*motion: int*) → None
Call the appropriate motion.

The `motions` dictionary will be consulted, and if the provided motion is found, then that function will be called.

This is the default event that is used by `pyglet.window.Window`.

Parameters **motion** – One of the motion constants from pyglet.window.key.

start_ambiances() → None
Start all the ambiances on this instance.

start_tracks() → None
Start all the tracks on this instance.

stop_ambiances() → None
Stop all the ambiances on this instance.

stop_tracks() → None
Stop all the tracks on this instance.

earwax.mixins module

Provides various mixin classes for used with other objects.

class earwax.mixins.**DismissibleMixin**(*dismissible: bool = True*)
Bases: `object`

Make any `Level` subclass dismissible.

Variables **dismissible** – Whether or not it should be possible to dismiss this level.

dismiss() → None
Dismiss the currently active level.

By default, when used by `earwax.Menu` and `earwax.Editor`, this method is called when the escape key is pressed, and only if `self.dismissible` evaluates to `True`.

The default implementation simply calls `pop_level()` on the attached `earwax.Game` instance, and announces the cancellation.

class earwax.mixins.**DumpLoadMixin**
Bases: `object`

A mixin that allows any object to be dumped to and loaded from a dictionary.

It is worth noting that only instance variables which have type hints (and thus end up in the `__annotations__` dictionary) will be dumped and loaded.

Also, any instance variables whose name starts with an underscore (_) will be ignored.

To dump an instance, use the *`dump()`* method, and to load, use the *`load()`* constructor.

The `__allowed_basic_types__` list holds all the types which will be dumped without any modification.

By default, the only collection types that are allowed are `list`, and `dict`.

If you wish to exclude attributes from being dumped or loaded, create a `__excluded_attributes__` list, and add all names there.

dump() → Dict[str, Any]
Dump this instance as a dictionary.

classmethod from_file(*f: TextIO*, **args*) → Any

Return an instance from a file object.

Parameters

- **f** – A file which has already been opened.
- **args** – Extra positional arguments to pass to the `load` constructor.

classmethod from_filename(*filename: pathlib.Path*, **args*) → Any

Load an instance from a filename.

Parameters **filename** – The path to load from.

get_dump_value(*type_: Type[CT_co], value: Any*) → Any

Get a value for dumping.

Parameters **value** – The value that is present on the instance.

classmethod get_load_value(*expected_type: Type[CT_co], value: Any*) → Any

Return a loaded value.

In the event that the dumped value represents a instance of `earwax.mixins.DumpLoadValue`, the dictionary must have been returned by *earwax.mixins.DumpLoadMixin.dump()*, so it contains both the dumped value, and the type annotation.

This prevents errors with Union types representing multiple subclasses.

If the type of the provided value is found in the `__allowed_basic_types__` list, it will be returned as-is. This is also true if the value is an enumeration value.

If the type of the provided value is `list`, then each element will be passed through this method and a list of the loaded values returned.

If the type of the value is `dict`, one of two things will occur:

- **If expected_type is also a dict, then the given value will have** its keys and values loaded with this function.
- **If expected_type is also a subclass of** *earwax.mixins.DumpLoadMixin*, then it will be loaded with that class's `load` method.
- If neither of these things are true, `RuntimeError` will be raised.

Parameters

- **expected_type** – The type from the `__annotations__` dictionary.
- **value** – The raw value to load.

classmethod load(*data: Dict[str, Any], *args*) → Any

Load and return an instance from the provided data.

It is worth noting that only keys that are also found in the `__annotations__` dictionary, and not found in the `__excluded_attribute_names__` list will be loaded. All others are ignored.

Parameters

- **data** – The data to load from.
- **args** – Extra positional arguments to pass to the constructor.

save(*filename: pathlib.Path*) → None

Write this object to the provided filename.

Parameters **filename** – The path to the file to dump to.

class earwax.mixins.**RegisterEventMixin**

Bases: object

Allow registering and binding events in one function.

register_and_bind(*func: EventType*) → EventType

Register and bind a new event.

This is the same as:

```
level.register_event_type('f')

@level.event
def f() -> None:
    pass
```

Parameters **func** – The function whose name will be registered, and which will be bound to this instance.

register_event(*func: EventType*) → str

Register an event type from a function.

This function uses `func.__name__` to register an event type, eliminating possible typos in event names.

Parameters **func** – The function whose name will be used.

class earwax.mixins.**TitleMixin**(*title: Union[str, TitleFunction]*)

Bases: object

Add a title to any `Level` subclass.

Variables **title** – The title of this instance.

If this value is a callable, it should return a string which will be used as the title.

get_title() → str

Return the proper title of this object.

If `self.title` is a callable, its return value will be returned.

earwax.networking module

Provides classes for networking.

exception earwax.networking.**AlreadyConnected**

Bases: *earwax.networking.NetworkingConnectionError*

Already connected.

Attempted to call `connect()` on an already connected `NetworkConnection` instance.

exception earwax.networking.**AlreadyConnecting**

Bases: *earwax.networking.NetworkingConnectionError*

Already connecting.

An attempt was made to call `connect()` on an `NetworkConnection` instance which is already attempting to connect.

class earwax.networking.**ConnectionStates**

Bases: enum.Enum

Various states that `NetworkConnection` classes can be in.

Variables

- **`not_connected`** – The connection's `connect()` method has not yet been called.
- **`connecting`** – The connection is still being established.
- **`connected`** – A connection has been established.
- **`disconnected`** – This connection is no longer connected (but was at some point).
- **`error`** – There was an error establishing a connection.

`connected = 2`

`connecting = 1`

`disconnected = 3`

`error = 4`

`not_connected = 0`

`class` `earwax.networking.`**`NetworkConnection`**

Bases: *`earwax.mixins.RegisterEventMixin`*

Represents a single outbound connection.

You can read data by providing an event handler for `on_data()`, and write data with the `send()` method.

Variables

- **`socket`** – The raw socket this instance uses for communication.
- **`state`** – The state this connection is in.

`close``()` → None

Close this connection.

Disconnect `self.socket`, and call `shutdown()` to clean up..

`connect``(`*hostname: str*, *port: int*`)` → None

Open a new connection.

Connect `self.socket` to the provided hostname and port.

Parameters

- **`hostname`** – The hostname to connect to.
- **`port`** – The port to connect on.

`on_connect``()` → None

Deal with the connection being opened.

This event is dispatched when text is first received from `self.socket`, since I've not found a better way to know when the socket is properly open.

`on_data``(`*data: bytes*`)` → None

Handle incoming data.

An event which is dispatched whenever data is received from `self.socket`.

`on_disconnect``()` → None

Handle the connection closing.

Dispatched when `self.socket` has disconnected.

A socket disconnect is defined by the socket in question receiving an empty string.

on_error(*e: Exception*) → None
Handle a connection error.

This event is dispatched when there is an error establishing a connection.

Parameters **e** – The exception that was raised.

poll(*dt: float*) → None
Check if any data has been received.

Poll `self.socket` for anything that has been received since the last time this function ran.

This function will be scheduled by `connect()`, and unscheduled by `shutdown()`, when no more data is received from the socket.

If this connection is not connected yet (I.E.: you called this function yourself), then `earwax.NotConnectedYet` will be raised.

send(*data: bytes*) → None
Send some data over this connection.

Sends some data to `self.socket`.

If this object is not connected yet, then `NotConnectedYet` will be raised.

Parameters **data** – The data to send to the socket.

Must end with `'\r\n'`.

shutdown() → None
Shutdown this server.

Unschedule `self.poll`, set `self.socket` to `None`, and reset `self.state` to `earwax.ConnectionStates.not_connected`.

exception earwax.networking.**NetworkingConnectionError**
Bases: `Exception`

Base class for connection errors.

exception earwax.networking.**NotConnectedYet**
Bases: *earwax.networking.NetworkingConnectionError*

Tried to send data on a connection which is not yet connected.

earwax.point module

Provides the Point class.

class earwax.point.**Point**(*x: T, y: T, z: T*)
Bases: `typing.Generic`

A point in 3d space.

angle_between(*other: earwax.point.Point*) → float
Return the angle between two points.

Parameters **other** – The other point to get the angle to.

coordinates
Return `self.x`, `self.y`, and `self.z` as a tuple.

copy() → earwax.point.Point[~T][T]

Copy this instance.

Returns a `Point` instance with duplicate `x` and `y` values.

directions_to(*other: earwax.point.Point*) → earwax.point.PointDirections

Return the direction between this point and `other`.

Parameters **other** – The point to get directions to.

distance_between(*other: earwax.point.Point*) → float

Return the distance between two points.

Parameters **other** – The point to measure the distance to.

floor() → earwax.point.Point[int][int]

Return a version of this object with both coordinates floored.

in_direction(*angle: float*, *distance: float = 1.0*) → earwax.point.Point[float][float]

Return the coordinates in the given direction.

Parameters

- **angle** – The direction of travel.
- **distance** – The distance to travel.

classmethod origin() → earwax.point.Point[int][int]

Return `Point(0, 0, 0)`.

class earwax.point.**PointDirections**

Bases: `enum.Enum`

Point directions enumeration.

Most of the possible directions between two `Point` instances.

There are no vertical directions defined, although they would be easy to include.

east = 3

here = 0

north = 1

northeast = 2

northwest = 8

south = 5

southeast = 4

southwest = 6

west = 7

earwax.pyglet module

A mock pyglet module.

This module exists to prevent ReadTheDocs from kicking off when docs are built.

earwax.reverb module

Reverb module.

class earwax.reverb.**Reverb**(*gain: float = 1.0, late_reflections_delay: float = 0.01, late_reflections_diffusion: float = 1.0, late_reflections_hf_reference: float = 500.0, late_reflections_hf_rolloff: float = 0.5, late_reflections_lf_reference: float = 200.0, late_reflections_lf_rolloff: float = 1.0, late_reflections_modulation_depth: float = 0.01, late_reflections_modulation_frequency: float = 0.5, mean_free_path: float = 0.02, t60: float = 1.0*)

Bases: `object`

A reverb preset.

This class can be used to make reverb presets, which you can then upgrade to full reverbs by way of the `make_reverb()` method.

make_reverb(*context: object*) → object

Return a synthizer reverb built from this object.

All the settings contained by this object will be present on the new reverb.

Parameters **context** – The synthizer context to use.

earwax.rumble_effects module

Provides various rumble effect classes.

Please note:

When we talk about a rumble *value*, we mean a value from `0.0` (nothing), to `1.0` (full on).

In reality, values on the lower end can barely be felt with some controllers.

class earwax.rumble_effects.**RumbleEffect**(*start_value: float, increase_interval: float, increase_value: float, peak_duration: float, peak_value: float, decrease_interval: float, decrease_value: float, end_value: float*)

Bases: `object`

A rumble effect.

Instances of this class create rumble "waves", with a start, a climb in effect to an eventual peak, then, after some time at the peak, a gradual drop back to stillness.

For example, you could have an effect that started at 0.5 (half power), then climbed in increments of 0.1 every 10th of a second to a peak value of 1.0 (full power), then stayed there for 1 second, before reducing back down to 0.7 (70% power), with 0.1 decrements every 0.2 seconds.

The code for this effect would be:

```
effect: RumbleEffect = RumbleEffect(
    0.5,  # start_value
    0.1,  # increase_interval
    0.1,  # increase_value
    1.,  # peak_duration
    1.0,  # peak_value
    0.2,  # decrease_interval
    0.1,  # decrease_value
```

(continues on next page)

(continued from previous page)

```
    0.7,   # end_value
)
```

The `start()` method returns an instance of `StaggeredPromise`. This gives you the ability to save your effect, then use it at will:

```
effect: RumbleEffect = RumbleEffect(
    0.2,   # start_value
    0.3,   # increase_interval
    0.1,   # increase_value
    1.5,   # peak_duration
    1.0,   # peak_value
    0.3,   # decrease_interval
    0.1,   # decrease_value
    0.1,   # end_value
)
# ...
promise: StaggeredPromise = effect.start(game, 0)
promise.run()
```

Variables

- **`start_value`** – The initial rumble value.
- **`increase_interval`** – How many seconds should elapse between each increase.
- **`increase_value`** – How much should be added to the rumble value each increase.
- **`peak_duration`** – How many seconds the `peak_value` rumble should be felt.
- **`peak_value`** – The highest rumble value this effect will achieve.
- **`decrease_interval`** – The number of seconds between decreases.
- **`decrease_value`** – How much should be subtracted from the rumble value each decrease.
- **`end_value`** – The last value that will be felt.

`start`(*game: Game*, *joystick: object*) → earwax.promises.staggered_promise.StaggeredPromise
Start this effect.

Parameters

- **`game`** – The game which will provide the `start_rumble()`, and `stop_rumble()` methods.
- **`joystick`** – The joystick to rumble.

class `earwax.rumble_effects.`**`RumbleSequence`**(*lines: List[earwax.rumble_effects.RumbleSequenceLine]*)
Bases: `object`

A sequence of rumbles.

Variables **`lines`** – A list of rumble lines that make up effect.

`start`(*game: Game*, *joystick: object*) → earwax.promises.staggered_promise.StaggeredPromise
Start this effect.

Parameters

- **game** – The game which will provide the `start_rumble()`, and `stop_rumble()` methods.
- **joystick** – The joystick to rumble.

class earwax.rumble_effects.**RumbleSequenceLine**(*power: float, duration: int, after: Optional[float]*)

Bases: `object`

A line of rumble.

This class should be used in conjunction with the `RumbleSequence` class.

Variables

- **power** – The power of the rumble.
- **duration** – The duration of the rumble.
- **after** – The time to wait before proceeding to the next line.

  If this value is `None`, then no time will elapse.

  Set this value to `None` for the last line in the sequence, to avoid the promise suspending unnecessarily.

earwax.sdl module

Provides function for working with sdl2.

exception earwax.sdl.**SdlError**

Bases: `Exception`

An error in SDL.

earwax.sdl.**maybe_raise**(*value: int*) → None

Possibly raise `SdlError`.

Parameters **value** – The value of an sdl function.

If this value is `-1`, then an error will be raised.

earwax.sdl.**sdl_raise**() → None

Raise the most recent SDL error.

earwax.sound module

Provides sound-related functions and classes.

exception earwax.sound.**AlreadyDestroyed**

Bases: *earwax.sound.SoundError*

This sound has already been destroyed.

class earwax.sound.**BufferCache**(*max_size: int*)

Bases: `object`

A cache for buffers.

Variables

- **max_size** – The maximum size (in bytes) the cache will be allowed to grow before pruning.

  For reference, 1 KB is `1024`, 1 MB is `1024 ** 2`, and 1 GB is `1024 ** 3`.
- **buffer_uris** – The URIs of the buffers that are loaded. Least recently used first.
- **buffers** – The loaded buffers.
- **current_size** – The current size of the cache.

destroy_all() → None
Destroy all the buffers cached by this instance.

get_buffer(*protocol: str*, *path: str*) → object
Load and return a Buffer instance.

Buffers are cached in the `buffers` dictionary, so if there is already a buffer with the given protocol and path, it will be returned. Otherwise, a new buffer will be created, and added to the dictionary:

```
cache: BufferCache = BufferCache(1024 ** 2 * 512)  # 512 MB max.
assert isinstance(
    cache.get_buffer('file', 'sound.wav'), synthizer.Buffer
)
# True.
# Now it is cached:
assert cache.get_buffer(
    'file', 'sound.wav'
) is cache.get_buffer(
    'file', 'sound.wav'
)
# True.
```

If getting a new buffer would grow the cache past the point of `max_size`, the least recently used buffer will be removed and destroyed.

It is not recommended that you destroy buffers yourself. Let the cache do that for you.

At present, both arguments are passed to `synthizer.Buffer.from_stream`.

Parameters

- **protocol** – One of the protocols supported by Synthizer.

  As far as I know, currently only `'file'` works.
- **path** – The path to whatever data your buffer will contain.

get_size(*buffer: object*) → int
Return the size of the provided buffer.

Parameters **buffer** – The buffer to get the size of.

get_uri(*protocol: str*, *path: str*) → str
Return a URI for the given protocol and path.

This meth is used by `get_buffer()`. :param protocol: The protocol to use.

Parameters **path** – The path to use.

pop_buffer() → object
Remove and return the least recently used buffer.

prune_buffers() → None
Prune old buffers.

This function will keep going, until either there is only ‘ buffer left, or `current_size` has shrunk to less than `max_size`.

class `earwax.sound.`**`BufferDirectory`**(*buffer_cache: earwax.sound.BufferCache, path: pathlib.Path, glob: Optional[str] = None, thread_pool: Optional[concurrent.futures._base.Executor] = None*)

Bases: `object`

An object which holds a directory of `synthizer.Buffer` instances.

For example:

```python
b: BufferDirectory = BufferDirectory(
    cache, Path('sounds/weapons/cannons'), glob='*.wav'
)
# Get a random cannon buffer:
print(b.random_buffer())
# Get a random fully qualified path from the directory.
print(b.random_path())
```

You can select single buffer instances from the `buffers` dictionary, or a random buffer with the `random_buffer()` method.

You can select single `Path` instances from the `paths` dictionary, or a random path with the `random_path()` method.

Variables

- **`cache`** – The buffer cache to use.
- **`path`** – The path to load audio files from.
- **`glob`** – The glob to use when loading files.
- **`buffers`** – A dictionary of of `filename: Buffer` pairs.
- **`paths`** – A dictionary of `filename: Path` pairs.

`buffers_default`() → Dict[str, object]

Return the default value.

Populates the `buffers` and `paths` dictionaries.

`random_buffer`() → object

Return a random buffer.

Returns a random buffer from `self.buffers`.

`random_path`() → pathlib.Path

Return a random path.

Returns a random path from `self.paths`.

exception `earwax.sound.`**`NoCache`**

Bases: *`earwax.sound.SoundManagerError`*

This sound manager was created with no cache.

class `earwax.sound.`**`Sound`**(*context: object, generator: object, buffer: Optional[object] = None, gain: float = 1.0, looping: bool = False, position: Union[float, earwax.point.Point, None] = None, reverb: Optional[object] = None, on_destroy: Optional[Callable[[Sound], None]] = None, on_finished: Optional[Callable[[Sound], None]] = None, on_looped: Optional[Callable[[Sound], None]] = None, keep_around: bool = NOTHING*)

Bases: `object`

The base class for all sounds.

Variables

- **`context`** – The synthizer context to connect to.
- **`generator`** – The sound generator.
- **`buffer`** – The buffer that feeds `generator`.

  If this value is `None`, then this sound is a stream.
- **`gain`** – The gain of the new sound.
- **`loop`** – Whether or not this sound should loop.
- **`position`** – The position of this sound.

  If this value is `None`, this sound will not be panned.

  If this value is an `earwax.Point` value, then this sound will be a 3d sound, and the position of its `source` will be set to the coordinates of the given point.

  If this value is a number, this sound will be panned in 2d, and the value will be a panning scalar, which should range between `-1.0` (hard left), and `1.0` (hard right).
- **`on_destroy`** – A function to be called when this sound is destroyed.
- **`on_finished`** – A function to be called when this sound has finished playing, and `looping` evaluates to `False`.

  The timing of this event should not be relied upon.
- **`on_looped`** – A function to be called each time this sound loops.

  The timing of this event should not be relied upon.
- **`keep_around`** – Whether or not this sound should be kept around when it has finished playing.

  If this value evaluates to `True`, it is the same as setting the `on_finished` attribute to `destroy()`.
- **`source`** – The synthizer source to play through.

`check_destroyed``()` → None

Do nothing if this sound has not yet been destroyed.

If it has been destroyed, `AlreadyDestroyed` will be raised.

`connect_reverb``(`*reverb: object*`)` → None

Connect a reverb to the source of this sound.

Parameters **`reverb`** – The reverb object to connect.

`destroy``()` → None

Destroy this sound.

This method will destroy the attached `generator` and `source`.

If this sound has already been destroyed, then `AlreadyDestroyed` will be raised.

`destroy_generator``()` → None

Destroy the `generator`.

This method will leave the `source` intact, and will raise `AlreadyDestroyed` if the generator is still valid.

`destroy_source``()` → None
Destroy the attached `source`.

If the source has already been destroyed, `AlreadyDestroyed` will be raised.

`destroyed`
Return whether or not this sound has been destroyed.

`disconnect_reverb``()` → None
Disconnect the connected `reverb` object.

`classmethod from_path``(`*context: object*, *buffer_cache: earwax.sound.BufferCache*, *path: pathlib.Path*, ***kwargs*`)` → earwax.sound.Sound
Create a sound that plays the given path.

Parameters

- **`context`** – The synthizer context to use.
- **`cache`** – The buffer cache to load buffers from.
- **`path`** – The path to play.

  If the given path is a directory, then a random file from that directory will be chosen.

Parm kwargs Extra keyword arguments to pass to the `Sound` constructor.

`classmethod from_stream``(`*context: object*, *protocol: str*, *path: str*, ***kwargs*`)` → earwax.sound.Sound
Create a sound that streams from the given arguments.

Parameters

- **`context`** – The synthizer context to use.
- **`protocol`** – The protocol argument for `synthizer.StreamingGenerator`.
- **`path`** – The path parameter for `synthizer.StreamingGenerator`.

`is_stream`
Return `True` if this sound is being streamed.

To determine whether or not a sound is being streamed, we check if `self.buffer` is `None`.

`pause``()` → None
Pause this sound.

`paused`
Return whether or not this sound is paused.

`play``()` → None
Resumes this sound after a call to `pause()`.

`reset_source``()` → object
Return an appropriate source.

`restart``()` → None
Start this sound playing from the beginning.

`set_gain``(`*gain: float*`)` → None
Change the gain of this sound.

Parameters **`gain`** – The new gain value.

set_looping(*looping: bool*) → None
Set whether or not this sound should loop.

Parameters **looping** – Whether or not to loop.

set_position(*position: Union[float, earwax.point.Point, None]*) → None
Change the position of this sound.

If the provided position is of a different type than the `current one`, then the underlying `source` object will need to changee. This will probably cause audio stuttering.

Parameters **position** – The new position.

exception `earwax.sound.`**SoundError**
Bases: `Exception`

The base exception for all sounds exceptions.

class `earwax.sound.`**SoundManager**(*context: object, buffer_cache: Optional[earwax.sound.BufferCache] = NOTHING, name: str = 'Untitled sound manager', default_gain: float = 1.0, default_looping: bool = False, default_position: Union[float, earwax.point.Point, None] = None, default_reverb: Optional[object] = None*)
Bases: `object`

An object to hold sounds.

Variables

- **context** – The synthizer context to use.
- **cache** – The buffer cache to get buffers from.
- **name** – An optional name to set this manager aside from other sound managers when debugging.
- **default_gain** – The default `gain` attribute for sounds created by this manager.
- **default_looping** – The default `looping` attribute for sounds created by this manager.
- **default_position** – The default `position` attribute for sounds created by this manager.
- **default_reverb** – The default `reverb` attribute for sounds created by this manager.
- **sounds** – A list of sounds that are playing.

destroy_all() → None
Destroy all the sounds associated with this manager.

play_path(*path: pathlib.Path, **kwargs*) → earwax.sound.Sound
Play a sound from a path.

The resulting sound will be added to `sounds` and returned.

Parameters

- **path** – The path to play.
- **kwargs** – Extra keyword arguments to pass to the constructor of `earwax.Sound`.

  This value will be updated by the `update_kwargs()` method.

play_stream(*protocol: str, path: str, **kwargs*) → earwax.sound.Sound
Stream a sound.

The resulting sound will be added to `sounds` and returned.

For full descriptions of the `protocol`, and `path` arguments, check the synthizer documentation for `StreamingGenerator`.

Parameters

- **`protocol`** – The protocol to use.
- **`path`** – The path to use.
- **`kwargs`** – Extra keyword arguments to pass to the constructor of the `earwax.Sound` class.

  This value will be updated by the `update_kwargs()` method.

`register_sound`(*sound: earwax.sound.Sound*) → None
Register a sound with this instance.

Parameters **`sound`** – The sound to register.

`remove_sound`(*sound: earwax.sound.Sound*) → None
Remove a sound from the `sounds` list.

Parameters **`sound`** – The sound that will be removed

`update_kwargs`(*kwargs: Dict[str, Any]*) → None
Update the passed kwargs with the defaults from this manager.

Parameters **`kwargs`** – The dictionary of keyword arguments to update.

The `setdefault` method will be used with each of the default values from this object..

exception `earwax.sound.`**`SoundManagerError`**
Bases: `Exception`

The base class for all sound manager errors.

earwax.speech module

Provides the tts object.

You can use this object to output speech through the currently active screen reader:

```python
from earwax import tts
tts.output('Hello, Earwax.')
tts.speak('Hello, speech.')
tts.braille('Hello, braille.')
```

NOTE: Since version 2020-10-11, Earwax uses Cytolk for its TTS needs.

In addition to this change, there is now an extra `speech <earwax.EarwaxConfig.speech` configuration section, which can be set to make the `output()` method behave how you'd like.

earwax.task module

Provides the Task class.

class `earwax.task.`**`Task`**(*interval: Callable[[], float], func: Callable[[float], None]*)
Bases: `object`

A repeating task.

This class can be used to perform a task at irregular intervals.

By using a function as the interval, you can make tasks more random.

Parameters

- **`interval`** – The function to determine the interval between task runs.
- **`func`** – The function to run as the task.
- **`running`** – Whether or not a task is running.

`start`(*immediately: bool = False*) → None

Start this task.

Schedules `func` to run after whatever interval is returned by `interval`.

Every time it runs, it will be rescheduled, until `stop()` is called.

Parameters **`immediately`** – If `True`, then `self.func` will run as soon as it has been scheduled.

`stop`() → None

Stop this task from running.

earwax.track module

Provides the Track class.

`class` `earwax.track.`**`Track`**(*protocol: str*, *path: str*, *track_type: earwax.track.TrackTypes*)

Bases: `object`

A looping sound or piece of music.

A track that plays while a `earwax.Level` object is top of the levels stack.

Variables

- **`protocol`** – The `protocol` argument to pass to `synthizer.StreamingGenerator``.`
- **`path`** – The `path` argument to pass to `synthizer.StreamingGenerator`.
- **`track_type`** – The type of this track.

  This value determines which sound manager an instance will be connected to.
- **`sound`** – The currently playing sound instance.

  This value is initialised as part of the `play()` method.

`classmethod` **`from_path`**(*path: pathlib.Path*, *type: earwax.track.TrackTypes*) → earwax.track.Track

Return a new instance from a path.

Parameters

- **`path`** – The path to build the track from.

  If this value is a directory, a random file will be selected.
- **`type`** – The type of the new track.

`play`(*manager: earwax.sound.SoundManager*, ***kwargs*) → None

Play this track on a loop.

Parameters

- **manager** – The sound manager to play through.
- **kwargs** – The extra keyword arguments to send to the given manager's `play_stream()` method.

stop() → None
Stop this track playing.

class earwax.track.**TrackTypes**
Bases: `enum.Enum`

The type of a `Track` instance.

Variables

- **ambiance** – An ambiance which will never moved, such as the background sound for a map.

  This type should not be confused with the `earwax.Ambiance` class, which describes an ambiance which can be moved around the sound field.

- **music** – A piece of background music.

ambiance = 0

music = 1

earwax.types module

Provides various type classes used by Earwax.

earwax.utils module

Provides various utility functions used by Earwax.

earwax.utils.**english_list**(*items: List[str], empty: str = 'Nothing', sep: str = ', ', and_: str = 'and '*) → str
Given a list of strings, returns a string representing them as a list.

For example:

```
english_list([]) == 'Nothing'
english_list(['bananas']) == 'bananas'
english_list(['apples', 'bananas']) == 'apples, and bananas'
english_list(
    ['apples', 'bananas', 'oranges']
) == 'apples, bananas, and oranges'
english_list(['tea', 'coffee'], and_='or ') == 'tea, or coffee'
```

Parameters

- **items** – The items to turn into a string.
- **empty** – The string to return if `items` is empty.
- **sep** – The string to separate list items with.
- **and** – The string to show before the last item in the list.

`earwax.utils.`**`format_timedelta`**(*td: datetime.timedelta*, **args*, ***kwargs*) → str

Given a timedelta `td`, return it as a human readable time.

For example:

```
td = timedelta(days=400, hours=2, seconds=3)
format_timedelta(
    td
) == '1 year, 1 month, 4 days, 2 hours, and 3 seconds'
```

Note: It is assumed that a month always contains 31 days.

Parameters

- **`td`** – The time delta to work with.
- **`args`** – The extra positional arguments to pass to *`english_list()`*.
- **`kwargs`** – The extra keyword arguments to pass onto *`english_list()`*.

`earwax.utils.`**`nearest_square`**(*n: int*, *allow_higher: bool = False*) → int

Given a number `n`, find the nearest square number.

If `allow_higher` evaluates to `True`, return the first square higher than `n`. Otherwise, return the last square below `n`.

For example:

```
nearest_square(5) == 2  # 2 * 2 == 4
nearest_square(24, allow_higher=True) == 5  # 5 * 5 == 25
nearest_square(16) == 4
nearest_square(16, allow_higher=True) == 4
```

Parameters **`n`** – The number whose nearest square should be returned.

`earwax.utils.`**`pluralise`**(*n: int*, *single: str*, *multiple: Optional[str] = None*) → str

If `n == 1`, return `single`. Otherwise return `multiple`.

If `multiple` is `None`, it will become `single + 's'`.

For example:

```
pluralise(1, 'axe') == 'axe'
pluralise(2, 'axe') == 'axes'
pluralise(1, 'person', multiple='people') == 'person'
pluralise(2, 'person', multiple='people') == 'people'
pluralise(0, 'person', multiple='people') == 'people'
```

Parameters

- **`n`** – The number of items we are dealing with.
- **`single`** – The name of the thing when there is only 1.
- **`multiple`** – The name of things when there are numbers other than 1.

`earwax.utils.`**`random_file`**(*path: pathlib.Path*) → pathlib.Path

Call recursively until a file is reached.

Parameters **`path`** – The path to start with.

earwax.vault_file module

Provides the VaultFile class.

exception `earwax.vault_file.`**`IncorrectVaultKey`**

Bases: `Exception`

The wrong key was given, and the file cannot be decrypted.

class `earwax.vault_file.`**`VaultFile`**(*entries: Dict[str, Union[bytes, List[bytes]]] = NOTHING*)

Bases: `object`

A class for restoring hidden files.

This class is used for loading files hidden by the `earwax vault` command.

Most of the time, you want to create instances with the `from_path()` constructor.

To add files, use the `add_path()` method.

Variables **`entries`** – The files which you are saving.

The format of this dictionary is `{label:  data}`, where `data` is the contents of the file you added.

Labels don't necessarily have to be the names of the files they represent. They can be whatever you like.

`add_path`(*p: Union[pathlib.Path, Generator[pathlib.Path, None, None]], label: Optional[str] = None*) → str

Add a file or files to this vault.

This method will add the contents of the given file to the `entries` dictionary, using the given label as the key.

Parameters

- **`p`** – The path to load.

  If the provided value is a generator, the resulting dictionary value will be a list of the contents of every file in that iterator.

  If the provided value is a directory, then the resulting dictionary value will be a list of every file (not subdirectory) in that directory.

- **`label`** – The label that will be given to this entry.

  This value will be the key in the `entries` dictionary.

  If `None` is provided, a string representation of the path will be used.

  If `None` is given, and the `p` is not a single `Path` instance, `RuntimeError` will be raised.

classmethod **`from_path`**(*filename: pathlib.Path, key: bytes*) → earwax.vault_file.VaultFile

Load a series of files and return a `VaultFile` instance.

Given a path to a data file, and the *correct* key, load a series of files and return a `VaultFile` instance.

If the key is invalid, `earwax.InvalidFaultKey` will be raised.

Parameters

- **`filename`** – The name of the file to load.

  This *must* be a data file, generated by a previous call to `earwax.VaultFile.save()`, not a yaml file as created by the `earwax vault new` command.

- **`key`** – The decryption key for the given file.

save (*filename: pathlib.Path*, *key: bytes*) → None
Save this instance's entries to a file.

Path filename The data file to save to.

The contents of this file will be encrypted with the given key, and will be binary.

Parameters **key** – The key to use to encrypt the data.

This key must either have been generated by `cryptography.fernet.Fernet.generate_key`, or be of the correct format.

earwax.walking_directions module

Provides the walking_directions dictionary.

earwax.yaml module

Makes the importing of yaml easier on systems that don't support CDumper.

`earwax.yaml.`**dump** (*data*, *stream=None*, *Dumper=<class 'yaml.dumper.Dumper'>*, ***kwds*)
Serialize a Python object into a YAML stream. If stream is None, return the produced string instead.

`earwax.yaml.`**load** (*stream*, *Loader=None*)
Parse the first YAML document in a stream and produce the corresponding Python object.

class `earwax.yaml.`**CDumper** (*stream*, *default_style=None*, *default_flow_style=False*, *canonical=None*, *indent=None*, *width=None*, *allow_unicode=None*, *line_break=None*, *encoding=None*, *explicit_start=None*, *explicit_end=None*, *version=None*, *tags=None*, *sort_keys=True*)
Bases: `yaml._yaml.CEmitter`, `yaml.serializer.Serializer`, `yaml.representer.Representer`, `yaml.resolver.Resolver`

class `earwax.yaml.`**CLoader** (*stream*)
Bases: `yaml._yaml.CParser`, `yaml.constructor.Constructor`, `yaml.resolver.Resolver`

9.1.3 Module contents

The Earwax game engine.

Earwax

This package is heavily inspired by Flutter.

Usage

- Begin with a `Game` object:

```
from earwax import Game, Level
g = Game()
```

- Create a level:

```
l = Level()
```

- Add actions to allow the player to do things:

```
@l.action(...)
def action():
    pass
```

- Create a Pyglet window:

```
from pyglet.window import Window
w = Window(caption='Earwax Game')
```

- Run the game you have created:

```
g.run(w)
```

There are ready made `Level` classes for creating `menus`, and `editors`.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index