Pyskool Documentation

Release 1.1

Richard Dymond

February 10, 2014

CHAPTER 1

What is Pyskool?

In 1985, Microsphere published Skool Daze, a game for the Sinclair ZX Spectrum. Later in the same year, the sequel Back to Skool was published.

Each game was based in a boys' school (though Back to Skool added a playground and a girls' school) and revolved around the antics of Eric, the hero. In Skool Daze, Eric must steal his report card from the school safe - the combination of which must be extracted from the teachers' brains using flashing shields or, in the case of the history teacher, post-hypnotic suggestion. In Back to Skool, Eric must get his report card back into the school safe, this time with the extra help provided by a water pistol, stinkbombs, a bike, mice, a frog and a girlfriend.

Pyskool is a re-implementation of these classic games in Python and Pygame, with the aim of making them easy to customise by editing a configuration file or - for more advanced customisation - writing some Python code.

The latest version of Pyskool can always be obtained from pyskool.ca.

1.1 Licence and copyrights

CHAPTER 2

Installing and running Pyskool

2.1 Requirements

Pyskool requires Python (version 2.6 or 2.7) and Pygame (version 1.7+).

On Linux/*BSD, Python and Pygame are available via the package management system. Python is in the *python* package on all systems; Pygame is in the *python-pygame* package on Debian-based distros and openSUSE, the *pygame* package on Fedora, the *devel/py-game* port on FreeBSD and NetBSD, and the *devel/pygame* port on OpenBSD.

Windows and Mac OS X users should take care to select the Pygame installer that matches the version of Python that is installed.

2.2 Installing Pyskool

Pyskool can be run from wherever the zip archive or tarball was unpacked - it does not need to be installed in any particular location. However, if you would like to install Pyskool as a Python package, you can do so by using the supplied `setup.py` script. After installation, the game launcher scripts (*skool_daze.py*, *back_to_skool.py* and the others) can be run from anywhere, instead of just the directory in which the Pyskool zip archive or tarball was unpacked.

2.2.1 Windows

To install Pyskool as a Python package on Windows, open a command prompt, change to the directory where Pyskool was unpacked, and run the following command:

```
> setup.py install
```

This should install the Pyskool game launcher scripts in *C:\Python2X\Scripts* (assuming you have installed Python in *C:\Python2X*), which means you can run them from anywhere (assuming you have added *C:\Python2X\Scripts* to the `Path` environment variable).

2.2.2 Linux/*BSD/Mac OS X

To install Pyskool as a Python package on Linux/*BSD/Mac OS X, open a terminal window, change to the directory where Pyskool was unpacked, and run the following command as root:

```
# ./setup.py install
```

This should install the Pyskool game launcher scripts in */usr/local/bin* (or some other suitable location in your `PATH`), which means you can run them from anywhere.

2.3 Running Pyskool

2.3.1 Windows

To run Pyskool in Skool Daze mode, double-click the *skool_daze.py* file in the Pyskool directory. To run Pyskool in Back to Skool mode, double-click *back_to_skool.py*.

If that doesn't work, try the command line. Open a command prompt, change to the Pyskool directory, and do:

```
> skool_daze.py
```

to run Pyskool in Skool Daze mode; or, to run Pyskool in Back to Skool mode:

```
> back_to_skool.py
```

2.3.2 Linux/*BSD/Mac OS X

To run Pyskool in Skool Daze mode, open a terminal window, change to the Pyskool directory, and do:

```
$ ./skool_daze.py
```

or, to run Pyskool in Back to Skool mode:

```
$ ./back_to_skool.py
```

2.4 Pyskool data files

When *skool_daze.py*, *back_to_skool.py* or one of the other game launcher scripts is executed, it looks for the following things:

- a file named *pyskool.ini* (the main ini file)
- a directory named *images*
- a directory named *sounds*
- a directory named *ini/<game_name>* (where *<game_name>* is *skool_daze*, *back_to_skool*, or whatever)

Each of these things must be present in one of the following directories in order for Pyskool to find it:

- the current working directory
- *$HOME/.pyskool*
- the directory containing the game launcher script
- */usr/share/pyskool*
- *$PACKAGE_DIR/data*

$HOME refers to the user's home directory. On Windows this is typically *C:\Documents and Settings\username*.

$PACKAGE_DIR refers to the directory in which the *pyskool* package is installed (as shown by the `--package-dir` command line option).

When you need a reminder of the locations that Pyskool searches for data files, run one of the game launcher scripts with the `--search-dirs` option.

If Pyskool doesn't start, run the game launcher script from the command line and read the diagnostic messages that are printed to the console for clues about what's going wrong.

When Pyskool is running, it will dump screenshots to, save games to, and load games from either *$HOME/.pyskool* (if it exists or can be created), or the current working directory.

2.5 Command line options

skool_daze.py, *back_to_skool.py* and the other game launcher scripts support the following command line options:

- `--version` - show the version number of Pyskool and exit
- `-h` or `--help` - show a summary of the available options
- `-c` or `--cheat` - enable cheat keys; overrides the *Cheat* setting in the *[GameConfig]* section
- `--create-images` or `--get-images` - create the images required by the game and exit
- `--create-ini` - create the ini files required by the game in *$HOME/.pyskool/ini/<game_name>* and exit
- `--create-sounds` - create the sound files required by the game in *$HOME/.pyskool/sounds* and exit
- `-i INIDIR` or `--inidir=INIDIR` - use ini files from a specified directory
- `-l SAVEFILE` or `--load=SAVEFILE` - load a previously saved game
- `--package-dir` - show the path to the pyskool package directory and exit
- `-q` or `--quick-start` - start the game quickly by skipping the scroll-skool-into-view and theme tune sequence; overrides the *QuickStart* setting in the *[GameConfig]* section
- `-r SAVEDIR` or `--load-last=SAVEDIR` - load the most recently saved game from the specified directory
- `-s SCALE` or `--scale=SCALE` - set the scale of the display; overrides the *Scale* setting in the *[ScreenConfig]* section
- `--search-dirs` - show the locations that Pyskool searches for data files and exit
- `--setup` - create the images, ini files and sound files required by the game in *$HOME/.pyskool* and exit

The `--create-images` option first looks for Skool Daze and Back to Skool tape or snapshot files by the following names in *$HOME/.pyskool*:

- *skool_daze.tzx*
- *skool_daze.sna*
- *skool_daze.z80*
- *skool_daze.szx*
- *back_to_skool.tzx*
- *back_to_skool.sna*
- *back_to_skool.z80*
- *back_to_skool.szx*

If no such files are found, TZX files are downloaded from one of the sources listed in *images.ini* and saved to *$HOME/.pyskool*. Then the required images are built from the tape or snapshot files and saved to the appropriate subdirectories under *$HOME/.pyskool/images/originalx1*.

CHAPTER 3

Playing Pyskool

3.1 Skool Daze mode

3.1.1 Original game instructions

In the role of our hero, Eric (or any other name you decide to call him and the rest of the cast), you know that inside the staffroom safe are kept the school reports. And, being Eric, you realise that you must at all costs remove your report before it comes to the attention of the Headmaster.

The combination to the safe consists of four letters, each master knowing one letter and the Headmaster's letter always coming first. To get hold of the combination, you first have to hit all the shields hanging on the school walls. Trouble is, this isn't as easy as it looks. Some of them can be hit by jumping up. Others are more difficult. You could try and hit a shield by bouncing a pellet off a master's head whilst he is sitting on the ground. Or, being Eric, you may decide to knock over one of the boys and, whilst he's flattened, clamber up on him so that you can jump higher.

OK. So all the shields are flashing wildly, disorientating the poor masters. Knock them over now and, before they can stop themselves, they'll reveal their letter of the code. All except for the history master, of course, who because of his great age and poor eyesight can't be trusted to remember. His letter has been implanted into his mind hypnotically. To make him reveal it, you must find out the year he was born (which, in case you were wondering, changes each game). Then, creep into a room before he gets there and, if the board is clean, write it on the blackboard. When he goes into that room and sees his birthdate he will, as if by post-hypnotic suggestion, give away his letter.

Now that you know all the letters of the combination, all you have to do is work out which order they go in. You know that the Headmaster's letter is always first, but as for the other three...you'll just have to try the various possibilities. Find a clean blackboard and write out a combination.

Rush back to the staffroom and jump up to reach the safe with your hand. If nothing happens, then the combination must be wrong, so you'd better find another clean blackboard and try a different one.

With the safe open, your troubles still aren't over, as the flashing shields are rather a giveaway. To stop them flashing, you now have to hit all of them again.

Done it? Congratulations! You are now allowed, along with all your friends, to move on to the next class at school. But remember, there will be reports at the end of this term...

3.1.2 School Rules

Boys shall attend lessons as shown in the timetable at the bottom of the screen. (Remember that because you cheated in the exams last year, you always go to the same lessons as the swot.)

Boys do not score points by attending lessons, but may be given lines if caught in the wrong place.

Boys who acquire over 10,000 lines shall be expelled immediately from the school.

Boys are not allowed to enter the staffroom or the Headmaster's study. Take care.

At playtime, boys are supposed to be playing and not in any of the classrooms.

Boys shall not hit their schoolmates.

Boys shall not fire catapults.

Boys are expected to walk quietly in the corridors - they are not for running or sitting in.

School dinners are compulsory.

Boys will be neat and polite at all times.

3.2 Back to Skool mode

3.2.1 Original game instructions

Having managed to steal your report during the last few skooldaze of term, you've spent the whole holiday forging teachers' signatures and handwriting to make yourself look like the brightest, sweetest natured, most helpful little soul ever to carry a satchel.

Now all you've got to do is get it back into the Head's safe...

A couple of years ago, your elder brother had exactly the same problem and has lent you his old copy of the School Rules, on which he's scribbled some notes in invisible ink. Holding the paper over a bonfire made from the swot's cap, you can just make out the following:

- Key to safe round nek of gurls hedmistris. She hates frogs.
- Put frog in cup in gurls kitchin so you can nock it down onto her hed. Remmember the bike.
- Frogs ar kept in loked siense lab stor room. Combernasion letters held by masters - get them drunk.
- Sherry in hedmistris's cubord - only unloked wen sheez shure all the boys ar back in skool.
- Fill water pistle by jumping up to botel. Put sherry in skool cups and use catopult to nock cups onto masters.
- Stink boms - Hed will open window if you drop one wen heez near.
- Cairtaker - if you nock him out with a conker by firing catopult from top window of skool, yool be able to get passed him.
- Bike - chaned to conker tree - 4 number combernasion has to be ritten on blackbord to relees it. Each master nose 1 number - soke them with water by nocking water-filled cups with catopult wilst they ar under.
- Water pistle and stink boms - hidden in desks. Remmember there ar desks in the gurls skool too.
- Water makes flours grow.
- Catching mice is fun. Letting them go in the gurls skool is even funier.
- If you get lots of lines to do, try being ekstra nice to your gurlfrend!
- Only the Hed can open the study dore - yool have to jump up to reech the safe.

3.3 Keys

The keys to move Eric around are:

- 'q' or up arrow - go up stairs, or continue walking in the same direction
- 'a' or down arrow - go down stairs, or continue walking in the same direction
- 'o' or left arrow - left
- 'p' or right arrow - right
- 'f' - fire catapult
- 'h' - hit
- 'j' - jump
- 's' - sit/stand
- 'w' - write on a blackboard (press Enter/Return to finish)

Eric always walks fast. In the original games Eric walked slow unless you held down the shift key; I always held down the shift key, so didn't feel motivated to include a 'slow mode'.

Other useful keys are:

- Escape - quit the game
- End - pause/resume
- Insert - take a screenshot
- F2 - save the game
- F6 - load the most recently saved game
- F12 - show/hide the menu

Screenshots are dumped to a BMP file (if using Pygame 1.7) or a PNG file (if using Pygame 1.8+).

Use the up and down arrow keys to move between items in the menu, and press Enter or the space bar to execute a menu item.

3.3.1 Back to Skool mode

Back to Skool mode also uses the following keys:

- 'c' - catch a mouse or frog
- 'd' - drop a stinkbomb
- 'g' - fire the water pistol
- 'm' - mount the bike
- 'o' - open a desk (while sitting at one)
- 'r' - release mice
- 't' - throw away the water pistol

To pedal the bike, use the left and right arrow keys (or 'o' and 'p'). Use the up arrow key (or 'q') to stand on the saddle, and the down arrow key (or 'a') to dismount. You can jump while standing on the saddle by pressing 'j' or the up arrow key (or 'q').

CHAPTER 4

Example customisations

So now you know everything there is to know about the ini files and command lists, you're ready to don your 'modding' hat and get customising. Right? Well, if not, you might want to follow the example customisations below to get a feel for what's possible.

4.1 Cursing CREAK

Maybe the simplest thing to customise is what the characters say. In this example we customise Mr Creak's sit down message (i.e. how he tells the kids to sit down when it's time to start a lesson).

To do this, find the `[SitDownMessages]` section in *skool_daze/messages.ini* and edit the line:

```
CREAK, BE QUIET AND SEATED YOU NASTY LITTLE BOYS
```

so that it reads:

```
CREAK, ARSES ON SEATS YOU SCUMMY LITTLE BUGGERS
```

(or whatever other polite request you'd like to see Mr Creak utter). Then run *skool_daze.py* and giggle like a schoolboy (or girl) as the profanities pour from the history teacher's mouth.

4.2 Blackboard blasphemy

An equally simple and giggle-inducing trick is to change what the characters write on the blackboards. In this example we'll modify Mr Withit's blackboard messages in Back to Skool.

Open up *back_to_skool/messages.ini* and find the `[BlackboardMessages WITHIT]` section. In there you will see the following messages:

```
ARTESIAN^WELLS
THE DOLDRUMS
TASTY^GEYSERS
THE GREEN^REVOLUTION
TREACLE^MINING
FROG FARMING
```

For fun, you can replace these messages with something more interesting, or add more messages (having only six to choose from makes Mr Withit a dull man). Note that the `^` character will be replaced with a newline.

Then run *back_to_skool.py* and smile with satisfaction as Mr Withit complies with your particular blackboard message whims.

4.3 800 LINES PERKINS

Did you ever think it was unfair that the little kids in Skool Daze (i.e. not Eric, the swot, the tearaway or the bully) never got lines? Eric would down a teacher with a catapult pellet, but any little kids in the vicinity could breeze past the teacher with complete impunity while Eric (or, if he was lucky, one of the other big kids) got slapped in the face with a bunch of lines.

Well this is Pyskool, and we can change all that. To make the little kids in Skool Daze potential lines recipients, go to the `[Characters]` section in *skool_daze/sprites.ini* and find the lines corresponding to the little boys:

```
BOY01, PERKINS, BOY, WALK0, -1, (43, 17), (1, 1), F
BOY02, GIBSON, BOY, WALK0, 1, (44, 17), (1, 1), F
BOY03, FANSHAW, BOY, WALK0, -1, (45, 17), (1, 1), F
...
```

The last field in each line (which contains `F` by default) is the character flags field. The character flag that turns a character into a potential lines recipient is `R`. So add that flag to each line, thus:

```
BOY01, PERKINS, BOY, WALK0, -1, (43, 17), (1, 1), FR
BOY02, GIBSON, BOY, WALK0, 1, (44, 17), (1, 1), FR
BOY03, FANSHAW, BOY, WALK0, -1, (45, 17), (1, 1), FR
...
```

Now run *skool_daze.py*, find a teacher milling about with a bunch of little kids, let rip with the catapult, and experience the satisfaction of seeing the hitherto nameless ones get their come-uppance.

4.4 Punch the pedagogue

The teachers in Skool Daze and Back to Skool - well, Mr Wacker and Mr Creak in particular - were always asking for a smack. Unfortunately, in the original games teachers were inexplicably impervious to Eric's pugilistic efforts. Eric could always whip out his catapult and send a teacher to the floor with a pellet, but it's not quite the same thing.

Anyway, with Pyskool, you get to change the rules. To make the teachers punchable with effect, open up *skool_daze/sprites.ini* or *back_to_skool/sprites.ini* and go to the `[Characters]` section. There you will find the lines corresponding to the teachers; in *skool_daze/sprites.ini* they look like this:

```
WACKER, MR WACKER/Sir, WACKER, WALK0, -1, (10, 17), (1, 0), ALPSTW
ROCKITT, MR ROCKITT/Sir, ROCKITT, WALK0, -1, (10, 17), (1, 0), ALPSTW
WITHIT, MR WITHIT/Sir, WITHIT, WALK0, -1, (10, 17), (1, 0), ALPSTW
CREAK, MR CREAK/Sir, CREAK, WALK0, -1, (10, 17), (1, 0), ALPSTW
```

The last field in each of these lines is the *flags* field (see *[Characters]*). To make a teacher punchable, we need to add the `F` flag. For example:

```
WACKER, MR WACKER/Sir, WACKER, WALK0, -1, (10, 17), (1, 0), AFLPSTW
```

Make the change for each teacher you'd like to see Eric (and, as a side effect, the bully too) be able to punch, and off you go and get your long-awaited revenge.

4.5 History in the Map Room

A somewhat more involved customisation is creating a new lesson. In this example we'll create a lesson where Mr Creak teaches Eric in the Map Room. In the original Skool Daze, Mr Creak never taught anywhere but in the Reading and White Rooms, so it'll be good for him to stretch his ageing legs and get on over to the Map Room.

4.5.1 Adding an entry to the timetable

First we are going to add an entry to the `[Timetable]` section. So, open up *skool_daze/lessons.ini*, head over to the `[Timetable]` section, and insert a new lesson ID or replace an existing one - preferably near the top, so you don't have to flick through too many lessons in Pyskool in order to test it. The top few lessons in the stock *lessons.ini* are:

```
[Timetable]
Playtime-4
Withit-MapRoom-2
RevisionLibrary-3
```

You could replace `Withit-MapRoom-2` with `Creak-MapRoom-1`:

```
[Timetable]
Playtime-4
Creak-MapRoom-1
RevisionLibrary-3
```

This means that the second lesson of the day will be the one with ID `Creak-MapRoom-1`. But that lesson doesn't exist yet, because we just made it up. So now it's time to create the lesson.

4.5.2 Creating the lesson

Now that the `[Timetable]` section contains a brand new lesson ID, we have to make sure there is a corresponding `[Lesson ...]` section. For this we're going to take a short cut. Since a lesson with Mr Creak in the Map Room is going to be almost the same as a lesson in the Map Room with any other teacher, we're going to find one such lesson, copy and paste it, and make the necessary modifications.

A good candidate for this copy/paste/modify plan is the lesson `Withit-MapRoom-1`, so find the section named `[Lesson Withit-MapRoom-1 WITHIT, MapRoom]`, copy and paste it somewhere else amid the `[Lesson ...]` sections, and rename it thus:

```
[Lesson Creak-MapRoom-1 CREAK, MapRoom]
BOY01, ReadingRoom-Boy
BOY02, WhiteRoom-Boy
BOY03, ReadingRoom-Boy
BOY04, WhiteRoom-Boy
BOY05, ReadingRoom-Boy
BOY06, WhiteRoom-Boy
BOY07, ExamRoom-Boy
BOY08, ReadingRoom-Boy
BOY09, ExamRoom-Boy
BOY10, ExamRoom-Boy
BOY11, MapRoom-Boy
WACKER, ExamRoom-Teacher
ROCKITT, WhiteRoom-Teacher
WITHIT, MapRoom-Teacher
CREAK, ReadingRoom-Teacher
TEARAWAY, ExamRoom-Tearaway
BULLY, MapRoom-Bully
SWOT, MapRoom-Swot
```

Now we're almost done. All that remains is to assign the appropriate command list to Mr Creak, and an alternative appropriate command list to Mr Withit. The simplest thing to do is switch their command lists round, thus:

```
WITHIT, ReadingRoom-Teacher
CREAK, MapRoom-Teacher
```

And that's it. Now run *skool_daze.py*, and give Mr Creak a round of applause as he makes it to the Map Room for the first time in his long career.

4.6 All aboard the Science Lab

Let's try our hand at a completely new lesson in Back to Skool this time. What about one where every boy and girl piles into the Science Lab with Mr Rockitt? That should be interesting.

4.6.1 Adding an entry to the timetable

You're an old hand at this now. Open up *back_to_skool/lessons.ini* and inspect the `[Timetable]` section:

```
[Timetable]
Playtime-5
Creak-YellowRoom-2
Assembly
```

Let's replace that `Creak-YellowRoom-2` entry with a carefully chosen unique ID for our new lesson:

```
[Timetable]
Playtime-5
Rockitt-ScienceLab-AllAboard
Assembly
```

Time to create the lesson itself.

4.6.2 Creating the lesson

We'll use the copy/paste/modify trick again, but this time there will be a lot more modifying to do. The lesson `Rockitt-ScienceLab-1` would be a good template to use, so find the lesson section named `[Lesson Rockitt-ScienceLab-1 ROCKITT, ScienceLab]`, copy and paste it somewhere else amid the lesson sections, and rename it thus:

```
[Lesson Rockitt-ScienceLab-AllAboard ROCKITT, ScienceLab]
```

The next step is to assign appropriate command lists to the characters. The appropriate command list for the little boys and girls is `ScienceLab-Boy` - don't be fooled by the `-Boy` suffix. So modify those command lists thus:

```
GIRL01, ScienceLab-Boy
GIRL02, ScienceLab-Boy
...
BOY01, ScienceLab-Boy
BOY02, ScienceLab-Boy
...
BOY10, ScienceLab-Boy
```

Now for the teachers. Mr Rockitt will obviously have to be in the Science Lab and the other teachers might as well just wander around, since they'll have nothing better to do:

```
WITHIT, Walkabout1-Teacher
ROCKITT, ScienceLab-Teacher
CREAK, Walkabout2-Teacher
TAKE, GirlsSkoolWalkabout-Teacher
```

Mr Wacker and Albert are fine as they are. Next, the main kids. They all need to pile into the Science Lab:

```
TEARAWAY, ScienceLab-Tearaway
BULLY, ScienceLab-Bully
SWOT, ScienceLab-Swot
HEROINE, ScienceLab-Boy
```

Now we're ready. Run *back_to_skool.py*, and watch the Science Lab fill to bursting point. Fun.

4.7 Where's the chalk?

So you've modified messages and lessons, but to be brutally honest, you haven't proved yourself as a Pyskool modder until you've created your own command list. Recall that a command list is a list of commands (!) that control a character during a lesson.

In this exercise we'll take the command list that controls the tearaway when he's on a blackboard-defacing spree in Skool Daze:

```
[CommandList WriteOnBoards-Tearaway]
GoTo, ExamRoomBlackboard:Middle
WriteOnBoardUnless, Dirty
GoTo, WhiteRoomBlackboard:Middle
WriteOnBoardUnless, Dirty
GoTo, ReadingRoomBlackboard:Middle
WriteOnBoardUnless, Dirty
SetControllingCommand, FireNowAndThen
GoToRandomLocation
WalkAround, 10
Restart
```

and turn it into a command list that leaves the tearaway frustrated by the global chalk shortage. One simple way to do this is to replace the three `WriteOnBoardUnless` commands with these commands:

```
Say, "Hey, where's the chalk?"
Say, "OMG, no chalk here, either!"
Say, "WTF? Has Mr Creak been eating the chalk or something?"
```

And for good measure we'll insert another `Say` command after `GoToRandomLocation`:

```
Say, "Anybody got any chalk?"
```

When these modifications are complete, the command list should look like this:

```
[CommandList WriteOnBoards-Tearaway]
GoTo, ExamRoomBlackboard:Middle
Say, "Hey, where's the chalk?"
GoTo, WhiteRoomBlackboard:Middle
Say, "OMG, no chalk here, either!"
GoTo, ReadingRoomBlackboard:Middle
Say, "WTF? Has Mr Creak been eating the chalk or something?"
SetControllingCommand, FireNowAndThen
GoToRandomLocation
Say, "Anybody got any chalk?"
WalkAround, 10
Restart
```

Now run *skool_daze.py* and watch as the hapless tearaway's blackboard-daubing career is dashed to the ground.

4.8 Ready-made customisations

Some ready-made customised ini files are distributed with Pyskool to demonstrate what's possible with the Pyskool engine. These customisations are described in the following sections.

4.8.1 Skool Daze Take Too

To play 'Skool Daze Take Too', double-click *skool_daze_take_too.py* or run it from the command line thus:

```
$ ./skool_daze_take_too.py
```

and say hello to Skool Daze's new philosophy teacher, who may look somewhat familiar.

4.8.2 Ezad Looks

To play 'Ezad Looks', double-click *ezad_looks.py* or run it from the command line thus:

```
$ ./ezad_looks.py
```

and prepare to feel a little disoriented for a while.

4.8.3 Back to Skool Daze

To play 'Back to Skool Daze', double-click *back_to_skool_daze.py* or run it from the command line thus:

```
$ ./back_to_skool_daze.py
```

and hit some shields for old times' sake.

CHAPTER 5

General info

5.1 Contact details

To make complaints about or suggest improvements to Pyskool, or to submit some other piece of constructive criticism, contact me (Richard Dymond) at <*rjdymond AT gmail.com*>, or leave a comment on the Pyskool blog.

To report bugs, please use the bug tracker.

5.2 TODO

Pyskool is now functionally complete, by which I mean that it does everything that the original Skool games do. (Except for a demo mode; if you think there's something else it should do but doesn't, let me know.) However, there are a few things left on my TODO list, the main ones being:

- Tidy up the code
- Optimise the sprite-drawing and screen update code
- Add a demo mode
- Write a *Command*-writing HOWTO (demonstrating how to add commands to Pyskool)

5.3 Bugs

No doubt there are bugs in Pyskool - and in this documentation - or ways it deviates unacceptably from the original games. Please report any bugs (reproducible crashes, especially) you find, and help to make Pyskool a solid and stable platform for developing new Skool-based games. If you can provide a saved game that demonstrates the bug shortly after being loaded, all the better.

5.4 Frequently asked questions

At the time of writing this, there are no frequently asked questions, or even any infrequently asked questions. So for now I'll fill this section with questions made up by me.

How does Pyskool differ from the original games?

Though the conversion of the original games to Python/Pygame is pretty faithful (I think), there are some differences, noted below.

General differences:

- More than one character can be talking at any given time
- Characters can talk while off-screen (so it's possible for the scrolling screen to reveal a character mid-sentence)
- Eric cannot walk slowly

In Skool Daze mode:

- Boys can find the back seat in the Reading Room

In Back to Skool mode:

- Eric can release mice anywhere (not just in the girls' skool)
- Eric can re-catch mice that he has released; well, why not?
- The frog is visible from the start of the game; I think the only reason it was hidden in the original game was a lack of RAM (the frog shares its character buffer with the mouse, and the mouse needs to be visible from the start)
- The 'conker' sound effect is played when Albert (instead of when Einstein or Angelface) is struck by a conker

Why Python (and Pygame)?

Because Python is an elegant, expressive, and excellent programming language. Plus it enables rapid development, which is good because I'm writing Pyskool in my limited spare time. Pygame's pretty good too. I don't know how else I'd do graphics with Python.

Why Skool Daze and Back to Skool?

If you need to ask, you probably shouldn't be here. Actually, what *are* you doing here? Go and play Jet Set Willy, or something.

CHAPTER 6

Changelog

6.1 1.1 (2013-12-01)

- Replaced all the sound files with high-quality (44.1kHz) versions
- Added hitting sound effects (`HIT0`, `HIT1`) to Skool Daze
- Added the `ALARM` sound effect ID (for when Albert is telling Mr Wacker that Eric's escaping)
- Screenshots are now saved to the *screenshots* directory by default
- Added the `--create-sounds` command line option (to create the sound files required by a game)
- Added the `--package-dir` command line option (for showing the path to the pyskool package directory)
- Added the `--search-dirs` command line option (for showing the locations that Pyskool searches for data files)
- Added the `--setup` command line option (to create the images, ini files and sound files required by a game)
- Added a second source for the Skool Daze TZX file to *images.ini*
- Removed the documentation sources from the Pyskool distribution (they can be obtained from GitHub)

6.2 1.0.1 (2012-12-07)

- Moved the man pages to section 6

6.3 1.0 (2012-12-03)

- Added the `--get-images` command line option (to download TZX files of Skool Daze and Back to Skool from sources listed in *images.ini* and extract images from them)
- Added the `--create-ini` command line option (to create the stock game ini files)
- Added the ability to switch between full-screen and windowed mode by pressing F11
- Man pages for the game launcher scripts are included in the *man* directory
- Fixed the audio latency that can occur when using Pygame 1.8+
- Fixed the bug that enables Eric to ride the bike past Albert when he has his arm raised
- Fixed the bug that makes Eric remain aloft after the knocked out kid he's standing on (near a staircase) has risen

- Fixed the bug in Back to Skool Daze that makes the shield on the shelf in the boys' skool turn into a cup when Eric goes onto the next year
- Fixed the *ezad_looks/mutables.png* image (each pair of shield/safe images was in the wrong order)

6.4 0.6 (2011-06-05)

- Pyskool can be installed as a Python package using `setup.py install`
- Changed the menu show/hide key from F10 (which activates the menu bar in Windows) to F12
- Added default key bindings to *pyskool.ini*
- Fixed the bug that enables Eric to kiss Hayley while she's sitting down
- Fixed the bug that makes Mr Wacker give Eric lines for being on the floor or not in skool while expelling him for jumping out of the top-floor window
- Fixed the bug that causes sprite graphics to lose their transparency when a game saved at one colour depth is loaded at a higher colour depth

6.5 0.5.4 (2011-03-15)

Fixed the bug that causes a crash when Eric tries to get on the bike.

6.6 0.5.3 (2010-12-16)

Fixed the bug that prevented a saved game from loading when using `GraphicsMode` 0 (hi-res colour).

6.7 0.5.2 (2010-11-03)

- Added a jumping sound effect to Skool Daze
- Fixed a graphic glitch in the girls' shoes

6.8 0.5.1 (2010-06-21)

Fixed the bug that causes a crash during a non-question-and-answer lesson when the teacher has returned to the blackboard after fetching the truant Eric.

6.9 0.5 (2010-06-08)

- Added an in-game menu
- Screen can be rescaled while Pyskool is running
- Key bindings are defined in *pyskool.ini*

6.10 0.4 (2010-05-28)

- Added the ability to save and load games
- The score box is drawn using labels defined in the *[MessageConfig]* section
- Added lesson box background images
- Added message box images (now the message boxes in Skool Daze mode look like those used in the original game)

6.11 0.3 (2010-05-18)

- Moved data that was embedded in the Python code into the ini files: there are now over 100 more parameters to tweak in the *[GameConfig]*, *[ScreenConfig]*, *[LessonConfig]*, *[TimetableConfig]*, *[TimingConfig]* and *[AnimationPhases ...]* sections, and extra character-controlling arguments to play with in the *WalkAround*, *MoveAboutUntil*, *MoveMouse*, *MoveFrog*, and *WatchForEric* commands (for example)
- Added utility scripts *createini.py* (generates ini files) and *extract-png.py* (extracts graphics from memory snapshots of the original skool games)
- Added documentation sources in *docs-src*

6.12 0.2.4 (2010-04-30)

Added the following features:

- 'Back to Skool Daze' example customisation
- Keyboard is checked during long sound effects (so you can pause or quit while the tune is playing, for example)
- Screenshots can be taken while the game is paused
- [Screen] section in the ini files
- API documentation

6.13 0.2.3 (2010-04-13)

Added the 'Ezad Looks' example customisation.

6.14 0.2.2 (2010-04-02)

Added the following features in Back to Skool mode:

- Eric is paralysed and expelled after jumping out of the top-floor window
- Albert alerts Mr Wacker if he spots Eric trying to escape
- Mr Wacker shadows Eric after being alerted by Albert
- Mr Creak and Mr Rockitt behave correctly during assembly
- Mr Withit does assembly duty

- Eric gets lines for not sitting down facing the stage during assembly
- Eric gets lines for standing on plants
- Miss Take chases Eric out of the girls' skool if she spots him there when it's not playtime

Also fixed the following bugs:

- Game crashes if Eric tries to sit back on the saddle of the bike after standing on it
- Eric gets lines for riding the bike in the playground
- Eric gets lines if spotted falling from a window
- Screen scrolls right every time Eric kisses Hayley

6.15 0.2.1 (2010-03-26)

Added the following features in Back to Skool mode:

- Eric can release mice
- The girls and Miss Take will jump up and down or stand on a chair if they spot a mouse nearby
- Eric can kiss (or try to kiss) Hayley
- Eric can open desks and collect the water pistol or stinkbombs
- Eric can drop stinkbombs
- Mr Wacker will open a nearby window if he smells a stinkbomb
- Eric can fire the water pistol
- Eric can fill the water pistol with sherry
- Eric can throw away the water pistol
- Cups can be filled with water or sherry
- Plants grow when watered
- Eric can stand on plant pots
- Eric is lifted by a growing plant
- Eric can step off a fully grown plant through an open window
- Eric can step off a fully grown plant over the skool gate
- Drops of water or sherry can be knocked out of a cup with a catapult pellet
- Teachers reveal bike combination digits when hit by a drop of water
- Eric can unchain the bike by writing the combination on a blackboard
- Eric can ride the bike
- Eric can stand on the saddle of the bike
- Eric can jump off the saddle of the bike
- Eric is launched over the closed skool gate if he hits it while standing on the saddle of the bike
- Teachers reveal storeroom combination letters when hit by a drop of sherry
- Eric can get the storeroom key (and hence the frog) by writing the combination on a blackboard

- Conker falls from the tree when hit by a catapult pellet
- Falling conker can knock people out
- Eric can place the frog in a cup
- Eric can get the safe key by knocking the frog from a cup onto Miss Take's head
- Eric can open the safe by jumping up to it when he has the key

Also fixed the following bugs:

- Game crashes if a character is chasing or looking for Eric while Eric is on a staircase or jumping
- Eric does not get lines if caught writing on a blackboard
- Eric gets lines for being in the assembly hall during non-assembly periods

6.16 0.2 (2010-03-16)

- Added mice and frogs and the ability to catch them
- Fixed glitches in the animatory state graphics (*sprites.png*)
- Added the *SHERRY* sound sample
- Added the *GameFps* and *ScrollFps* configuration parameters

Also fixed the following bugs:

- Game crashes if you press 'Delete' while writing on a blackboard
- If a little boy talks to ERIC while he's writing on a blackboard, pressing 'U' has no effect
- During dinner, the teacher on duty keeps giving Eric lines for not finding a seat

6.17 0.1.2 (2009-07-22)

Fixed bug in Skool Daze mode where shields stay flashing after Eric's been expelled.

6.18 0.1.1 (2009-04-29)

Fixed bug where Eric gets trapped in his seat if he's knocked out of it by a catapult pellet and then tries to stand up.

6.19 0.1 (2008-11-12)

- Eric is expelled after exceeding the lines limit
- The swot tells tales
- Teachers track down Eric if he tries to skip class

In Skool Daze mode:

- Special playtimes have been implemented
- Teachers give lines for all possible infractions

- All commands required in Skool Daze mode have been implemented

6.20 0.0.4 (2008-10-24)

- Eric can write on blackboards
- Improved keyboard responsiveness
- Added ready-made example customisation: Skool Daze Take Too

In Skool Daze mode:

- Teachers reveal safe combination letters when all shields are flashing
- Eric can open the safe after writing the combination code on a blackboard
- Eric can unflash all the shields after opening the safe

6.21 0.0.3 (2008-10-08)

- Sound effects and tunes
- Teachers give lines for some infractions
- Eric can jump (into the air and onto other kids, too)
- Eric can make shields flash

6.22 0.0.2 (2008-09-23)

- Added `--scale` and `--ini` command line options
- Bully can knock people out
- Tearaway can fire catapult pellets
- Eric can do these things too
- Tearaway writes on the blackboards
- Implemented several previously unimplemented commands

6.23 0.0.1 (2008-09-09)

Initial public release.

CHAPTER 7

Technical reference

7.1 Graphics

The stock Pyskool graphics are stored in PNG files in subdirectories named *back_to_skool*, *common* and *skool_daze* under *images/originalx1*. The PNG files are:

- *bubble.png* - speech bubble and lip
- *font.png* - the skool font
- *inventory.png* - mouse, frog, water pistol etc. (Back to Skool only)
- *lesson_box.png* - the lesson box background
- *logo.png* - the, er, logo
- *message_box.png* - the message box background
- *mutables_ink.png* - doors, windows, shields, safe etc. (ink colours only)
- *mutables_paper.png* - doors, windows, shields, safe etc. (paper colours only)
- *mutables.png* - doors, windows, shields, safe etc. (full colour)
- *scorebox.png* - the score/lines/hi-score box background
- *skool_ink.png* - the skool (ink colours only)
- *skool_paper.png* - the skool (paper colours only)
- *skool.png* - the skool (full colour)
- *sprites.png* - the characters in various 'animatory states'

These images were extracted straight from memory snapshots of Skool Daze and Back to Skool, and are therefore identical to the graphics in the original games (hence the *original* prefix in the directory name), except for minor glitches that have been fixed. (See Skool Daze graphic glitches and Back to Skool graphic glitches.)

The **_ink.png* and **_paper.png* files are used in `GraphicsMode` 1 (see *[GameConfig]*) in order to emulate the Spectrum display, which was restricted to two colours ('ink' and 'paper') per 8x8-pixel block.

sprites.png is an 8x16 array of sprites for the characters in the game. These sprites are all facing left, and are flipped to obtain the corresponding right-facing sprites.

Any of these images can be customised using your favourite image editor.

Pyskool performs the following steps to determine the base directory for graphics to use in the game:

- Collect the values of `ImageSet` and `Scale` from the *[GameConfig]* and *[ScreenConfig]* sections of the config file
- Look for the directory *images/<ImageSet>x<Scale>*
- Use images from that directory if it exists, or...
- ...use images from *images/<ImageSet>x1* and scale them up

The actual image files used from the base directory are defined in the *[Images]* section.

If you wanted to create your own hi-res graphics at 2x the original Spectrum size, you could place them under a base directory called *images/Customx2* and use the following parameter value in the *[GameConfig]* section:

```
ImageSet, Custom
```

and the following parameter values in the *[ScreenConfig]* section:

```
Scale, 2
GraphicsMode, 0
```

7.2 Main ini file

The main ini file - *pyskool.ini* - defines key bindings and the appearance and content of the game menus. Each section is described below.

7.2.1 [Keys]

The `Keys` section defines the key bindings. Each line in the section has the form:

```
actionId, key1, key2, ...
```

where `actionId` is the identifier of the action to bind to the keys `key1`, `key2` and so on. Any number of keys may be bound to an action.

Pyskool recognises the following action identifiers for moving Eric:

- `LEFT` - move left
- `RIGHT` - move right
- `UP` - move up
- `DOWN` - move down
- `SIT_STAND` - sit down or stand up
- `OPEN_DESK` - open a desk
- `FIRE_CATAPULT` - fire the catapult
- `FIRE_WATER_PISTOL` - fire the water pistol
- `DROP_STINKBOMB` - drop a stinkbomb
- `HIT` - throw a punch
- `JUMP` - jump
- `WRITE` - start writing on a blackboard
- `ENTER` - finish writing on a blackboard

- `CATCH` - try to catch a mouse or frog
- `UNDERSTOOD` - acknowledge understanding of a message
- `MOUNT_BIKE` - mount the bike
- `DUMP_WATER_PISTOL` - throw away the water pistol
- `RELEASE_MICE` - release some mice
- `KISS` - try to kiss someone

In addition, Pyskool recognises the following identifiers for actions not related to moving Eric:

- `QUIT` - quit Pyskool
- `FULL_SCREEN` - toggle full-screen mode
- `PAUSE` - pause the game
- `SCREENSHOT` - take a screenshot
- `SAVE` - save the game
- `LOAD` - load the most recently saved game
- `MENU` - show the menu
- `MENU_EXIT` - hide the menu and resume the game
- `MENU_PREV` - move to the previous item in the menu
- `MENU_NEXT` - move to the next item in the menu
- `MENU_EXEC` - execute the selected menu item

Pygame uses keyboard constants to identify keys; a full list of those constants can be found in the pygame documentation. The key names (`key1`, `key2` etc.) declared in a line of the `Keys` section should match the names of the Pygame keyboard constants, but with the `K_` prefix removed.

7.2.2 [Menu ...]

Each `Menu ...` section defines a menu and its appearance. The section name has the form:

```
Menu menuId
```

where `menuId` is a unique identifier for the menu. The section may contain the following configuration parameters:

- `Alpha` - the transparency of the menu (0=fully transparent, 255=fully opaque)
- `Highlight` - the background colour of the selected menu item
- `Ink` - the ink colour to use for the title, menu items and status bar
- `Paper` - the main background colour
- `StatusPaper` - the background colour of the status bar
- `Title` - the menu title
- `TitlePaper` - the background colour of the title bar
- `Width` - the width of the menu (as a fraction of the screen width)

At the moment, *pyskool.ini* contains the definition for only one menu, whose unique ID must be `Main`.

7.2.3 [MenuItems ...]

Each `MenuItems ...` section defines the menu items for a menu. The section name has the form:

```
MenuItems menuId
```

where `menuId` is the unique identifier of the menu (defined by a *[Menu ...]* section).

Each line in the section has the form:

```
operation, text
```

where:

- `operation` is the unique ID of the operation to which the menu item is bound
- `text` is the text of the menu item

The operation IDs recognised by Pyskool are:

- `LOAD` - load the most recently saved game
- `QUIT` - quit Pyskool
- `RESUME` - hide the menu and resume the game
- `SAVE` - save the game
- `SCALE_DOWN` - decrease the scale factor by 1
- `SCALE_UP` - increase the scale factor by 1

7.3 Game ini files

The game ini files determine many aspects of the game, such as the names of the characters, the order of the lessons, and what characters do during lessons.

By default, the ini files for a particular game are arranged in one of the subdirectories of the *ini* directory thus:

- *command_lists.ini* - *command lists*
- *config.ini* - configuration parameters
- *font.ini* - font character bitmap descriptions (see *[Font]*)
- *lessons.ini* - the *main timetable* and *lessons*
- *messages.ini* - messages
- *skool.ini* - *walls*, *floors* and other parts of the skool
- *sprites.ini* - sprite and *character* definitions

However, this arrangement is quite arbitrary; Pyskool will read every file with a *.ini* suffix in the subdirectory regardless of its name or contents. So you could, if you wish, concatenate all the ini files into one large ini file, and Pyskool will still work.

What follows is a description of every section of the ini files. Armed with this knowledge, and by consulting the *command reference* where necessary, you'll be able to start mucking around with how Pyskool works and how the game characters behave.

If you can't be bothered to read any of this and instead just want to get your modding hands dirty right now, head over to *Example customisations*.

If you're even lazier than that, head over to the *ready-made customisations* that are distributed with Pyskool.

7.3.1 [AnimationPhases ...]

The `AnimationPhases ...` section names take the form:

```
AnimationPhases phaseSetId
```

where `phaseSetId` is a descriptive unique ID for the list of animation phases that follows.

The format of an animation phase depends on who uses the phase set.

Each phase in the phase sets used by the frog (`FrogTurnRound`, `FrogShortHop` and `FrogLongHop`) looks like this:

```
animatoryState, xInc, directionChange
```

where:

- `animatoryState` is the ID of the animatory state
- `xInc` is the x-coordinate increment
- `directionChange` is the direction multiplier (-1 to change direction, 1 to not)

Each phase in the phase sets used by Eric (`DescentMiddleWindow`, `DescentUpperWindow`, `ClimbSkoolGate` and `FlyOverSkoolGate`) looks like this:

```
xInc, yInc, animatoryState
```

where:

- `xInc` is the x-coordinate increment
- `yInc` is the y-coordinate increment
- `animatoryState` is the ID of Eric's animatory state

Each phase in the phase set used by the stream of water or sherry fired from a water pistol (`Water`) looks like this:

```
animatoryState, xInc, yInc, hit
```

where:

- `animatoryState` is the ID of the animatory state
- `xInc` is the x-coordinate increment
- `yInc` is the y-coordinate increment
- `hit` is 0 if the water cannot hit anything, 1 if it can hit a cup, or 2 if it can hit a plant or the ground in this phase

Each phase in the phase set used by a stinkbomb when dropped (`Stinkbomb`) looks like this:

```
animatoryState, direction
```

where:

- `animatoryState` is the ID of the animatory state
- `direction` is the direction (-1 for left, 1 for right)

7.3.2 [AssemblyMessages]

The `AssemblyMessages` section contains all the information required to build a message used by the headmaster during assembly in Back to Skool.

There are two types of entry in this section. The first type of entry is the message template entry:

```
MESSAGE, assemblyMessageTemplate
```

which defines the template for an assembly message. The section can contain one or more message templates.

The second type of entry is the macro replacement entry:

```
MACRO, text
```

where `MACRO` is the name of a macro that appears in a message template (prefixed by `$`), and `text` is the text to which the macro should expand. Multiple macro replacement entries may be defined for any given macro. When an assembly message is created, the message template is chosen at random, and the macro replacements are chosen at random.

In the stock Back to Skool ini files in Pyskool, there is only one assembly message template, which contains two macros (`$VERB` and `$NOUN`).

7.3.3 [Bike]

The `Bike` section contains a single line of the form:

```
bikeId, spriteGroupId, animatoryState, unchainXY, commandListId, topLeft, size, coords, moveDelay, pe
```

where:

- `bikeId` is the bike's ID
- `spriteGroupId` is the ID of the *sprite group* to use for the bike
- `animatoryState` is the bike's initial animatory state
- `unchainXY` is the bike's initial coordinates (in *(x, y)* form) after being unchained
- `commandListId` is the unique ID of the *command list* that the bike will use
- `topLeft` is the coordinates (in *(x, y)* form) of the top left of the image of the base of the tree with no bike attached
- `size` is the size of the image (in *(width, height)* form)
- `coords` are the coordinates (in *(x, y)* form) of the mutable image in the play area
- `moveDelay` is the delay between consecutive movements of the bike when wheeling along or being pedalled (the higher the number, the slower the bike will go)
- `pedalMomentum` is the momentum increment when the bike is pedalled
- `maxMomentum` is the maximum momentum the bike can have

The bike images can be found in *mutables.png* (or *mutables_ink.png* and *mutables_paper.png* if `GraphicsMode` is 1 - see *[ScreenConfig]*). *mutables.png* is arranged so that the image of the bike attached to the base of the tree is at *(x + width, y)*, where *(x, y)* are the coordinates of the image of the base of the tree with no bike attached. These two images are the same size.

7.3.4 [BlackboardMessages ...]

The `[BlackboardMessages ...]` section names take the form:

```
BlackboardMessages characterId
```

where `characterId` is the unique ID of a character (see *[Characters]*).

Each `BlackboardMessages` section contains a list of messages (one per line) that may be written on a blackboard by the character whose ID is `characterId`.

There are two special characters used in blackboard messages: `^` and `$`. `^` represents the newline character (as defined by the `Newline` configuration parameter in the *[MessageConfig]* section). `$` is used to prefix the unique ID of a character, as in:

```
TEARAWAY, i hate^$WACKER
```

where `$WACKER` will be replaced by whatever name has been given to the character whose unique ID is `WACKER`.

If no blackboard messages are defined for a particular character, that character will never write on a blackboard.

7.3.5 [Blackboards]

The `Blackboards` section defines the blackboards in the classrooms. Each line has the form:

```
roomId, topLeft, size, chalk
```

where:

- `roomId` is the classroom's unique ID (see *[Rooms]*)
- `topLeft` is the coordinates of the top-left of the blackboard
- `size` is the size (width, height) of the blackboard
- `chalk` is the chalk colour (as an RGB triplet) to use when writing on the blackboard

In the stock Pyskool ini files, `chalk` is set to (255, 255, 255) - bright white - which coincides with the transparent colour used in the skool ink image (see `SkoolInkKey` in the *[ScreenConfig]* section). This means that in graphics mode 1 (see `GraphicsMode` in the *[ScreenConfig]* section), anything written on the board will take on the background (paper) colour, which is how blackboards worked in the original games.

7.3.6 [CatapultPellets]

Each line in the `CatapultPellets` section has the form:

```
characterId, pelletId, spriteGroupId, commandListId, range, hitZone, hitXY
```

where:

- `characterId` is the unique ID of the catapult-wielding character
- `pelletId` is the unique ID of the catapult pellet
- `spriteGroupId` is the ID of the *sprite group* to use for the pellet
- `commandListId` is the unique ID of the *command list* that the pellet will use
- `range` is the distance the pellet will fly after being launched
- `hitZone` is the size of the interval at the end of the pellet's flight where it can knock a character over
- `hitXY` is the coordinates of the pellet within its sprite (used for collision detection)

Each character whose unique ID appears in this section will be fitted out with a catapult. In the stock Pyskool this will be Eric and the tearaway - the only characters with catapult-firing sprites.

7.3.7 [Chairs]

The `Chairs` section contains one line for each classroom in the skool. Each line has the form:

```
roomId, x1, x2, x3..., xN
```

where:

- `roomId` is the classroom's unique ID (see *[Rooms]*)
- `x1`, `x2`, `x3` and so on are the x-coordinates of the chairs in the classroom

The order in which the x-coordinates are listed is significant: `x1` should correspond to the 'front seat' and `xN` should correspond to the 'back seat'. So if `x1` < `xN`, characters will sit down facing left; if `x1` > `xN`, characters will sit down facing right.

7.3.8 [Characters]

Each line in the `Characters` section has the form:

```
characterId, name[/title], spriteGroupId, animatoryState, direction, (x, y), headXY, flags
```

and corresponds to a single character, where:

- `characterId` is the character's unique ID (which should be alphanumeric and is used to refer to the character in other parts of the ini file)
- `name` is the character's name (as displayed in the game), and `title` (if supplied) is the name used by the swot to address the character
- `spriteGroupId` is the ID of the *sprite group* to use for the character
- `animatoryState` is the character's initial animatory state
- `direction` is the character's initial direction (-1 for left, 1 for right)
- `(x, y)` are the character's initial coordinates
- `headXY` are the coordinates of the character's head within his sprite when he's standing upright (used for collision detection)
- `flags` is a string of flags defining the character's abilities and vulnerabilities

Recognised flags and their meanings are:

- `A` - is an adult
- `B` - belongs in the boys' skool
- `C` - can be knocked over by a conker (see also `Z`)
- `D` - can open doors and windows
- `F` - can be knocked out by a fist
- `G` - belongs in the girls' skool
- `K` - holds the key to the safe
- `L` - can give lines
- `M` - is scared of mice
- `N` - can smell stinkbombs (and will open a nearby window if possible)
- `P` - can be knocked over by a catapult pellet

- `R` - can receive lines
- `S` - holds a safe combination letter
- `T` - can be tripped up by a stampeding kid (see *TripPeopleUp*)
- `U` - lines received by this character will be added to Eric's total
- `V` - lines received by this character will be added to Eric's score
- `W` - usually walks (unlike kids who sometimes run)
- `X` - holds a bike combination digit
- `Y` - holds a storeroom door combination letter
- `Z` - will be temporarily paralysed if struck by a falling conker (see also `C`)

7.3.9 [CommandList ...]

The `[CommandList ...]` section names take the form:

```
CommandList commandListId
```

where `commandListId` is a descriptive unique ID for the list of commands that follows. These unique IDs are used in the *[Lesson ...]* sections: for each type of lesson there is, every character is assigned a single command list to follow.

One example of a command list is:

```
[CommandList Walkabout1-Wacker]
GoTo, HeadsStudy:Window
GoToRandomLocation
Restart
```

This command list is used occasionally by the headmaster; it makes him repeatedly go to one of his *random locations* and then back to his study.

Each line in a command list contains the command name followed by a comma-separated list of arguments. See the *command reference* for more details on the commands that may be used to control a character.

7.3.10 [Conker]

The `Conker` section defines a conker (as knocked out of the tree by a catapult pellet). It contains a single line of the form:

```
objectId, spriteGroupId, commandListId, minX, maxX, minY, maxY, hitXY
```

where:

- `objectId` is a unique ID for the conker
- `spriteGroupId` is the ID of the *sprite group* to use for the conker
- `commandListId` is the unique ID of the *command list* that the conker will use when knocked out of the tree
- `minX`, `maxX`, `minY` and `maxY` define the rectangle inside the tree that contains the conker; a pellet that hits a spot in that rectangle will cause a conker to fall
- `hitXY` is the coordinates of the conker within its sprite (used for collision detection)

7.3.11 [Cups]

The `Cups` section contains information about cups (of which there are four in Back to Skool, and none in Skool Daze). Each line describes a single cup, and has the form:

```
cupId, emptyTopLeft, size, coords
```

where:

- `cupId` is the unique ID of the cup
- `emptyTopLeft` is the coordinates (in *(x, y)* form) of the top left of the image of the cup when empty
- `size` is the size of the image (in *(width, height)* form)
- `coords` are the coordinates (in *(x, y)* form) of the cup in the skool

The cup images can be found in *mutables.png* (or *mutables_ink.png* and *mutables_paper.png* if `GraphicsMode` is 1 - see *[ScreenConfig]*). *mutables.png* is arranged so that the image of a cup when it contains water is at *(x + width, y)*, and the image of a cup when it contains sherry is at *(x + 2 * width, y)* (where *(x, y)* are the coordinates of the image of the cup when empty). The three images for any given cup are the same size.

7.3.12 [DeskLid]

The `DeskLid` section contains a single line of the form:

```
deskLidId, spriteGroupId, commandListId, xOffset
```

where:

- `deskLidId` is the unique ID of the desk lid
- `spriteGroupId` is the ID of the *sprite group* to use for the desk lid when raised
- `commandListId` is the unique ID of the *command list* that the desk lid will use
- `xOffset` - the offset (relative to the desk being opened) at which the desk lid should be displayed

7.3.13 [Desks]

Each line in the `Desks` section has the form:

```
roomId, x1, x2...
```

where

- `roomId` is a classroom's unique ID (see *[Rooms]*)
- `x1`, `x2` and so on are the x-coordinates of the desks in the classroom (which should be a subset of the x-coordinates of the chairs in the classroom - see *[Chairs]*)

Any chair that is in a room and at an x-coordinate that appears in the `Desks` section will be fitted out with a desk lid that can be raised (see *[DeskLid]*).

7.3.14 [Doors]

The `Doors` section contains details of the doors in the game. Each line has the form:

```
doorId, x, bottomY, topY, initiallyShut, autoShutDelay, shutTopLeft, size, coords[, climb[, fly]]
```

where:

- `doorId` is the door's unique ID
- `x` is the door's x-coordinate
- `bottomY` and `topY` are the y-coordinates of the bottom and top of the door
- `initiallyShut` is `Y` if the door should be shut when the game starts
- `autoShutDelay` is the delay before the door swings shut automatically; if zero or negative, the door will not shut automatically
- `shutTopLeft` is the coordinates (in *(x, y)* form) of the top left of the image of the door when shut
- `size` is the size of the image (in *(width, height)* form)
- `coords` are the coordinates (in *(x, y)* form) of the door in the skool
- `climb` is the ID of the sequence of *animation phases* to use for Eric if he climbs over the door when it's shut; if not defined, Eric will not be able to climb over the door
- `fly` is the ID of the sequence of *animation phases* to use for Eric if he flies over the door after hitting it while standing on the saddle of the bike; if not defined, Eric will not be able to fly over the door

The door images can be found in *mutables.png* (or *mutables_ink.png* and *mutables_paper.png* if `GraphicsMode` is 1 - see *[ScreenConfig]*). *mutables.png* is arranged so that the image of a door when open is at *(x + width, y)*, where *(x, y)* are the coordinates of the image of the same door/window when shut. The open/shut images for any given door are the same size.

7.3.15 [Eric]

The `Eric` section describes our hero, Eric. It contains a single line in the format:

```
characterId, name, spriteGroupId, animatoryState, direction, (x, y), headXY, flags[, bendOverHandXY]
```

where:

- `characterId` is Eric's unique ID (which should be alphanumeric)
- `name` is Eric's name
- `spriteGroupId` is the ID of the *sprite group* to use for Eric
- `animatoryState` is Eric's initial animatory state
- `direction` is Eric's initial direction (-1 for left, 1 for right)
- `(x, y)` are Eric's initial coordinates
- `headXY` are the coordinates of Eric's head within his sprite when he's standing upright (used for collision detection)
- `flags` is a string of flags defining Eric's abilities and vulnerabilities
- `bendOverHandXY` is the coordinates of Eric's hand within his left-facing *BENDING_OVER* sprite (used to determine where a mouse or frog should be when caught or released)

For a description of the available flags, see *[Characters]*.

7.3.16 [Font]

The `Font` section is used to determine the location and size of the font character bitmaps in the *font.png* graphic. Each line has the form:

```
"char", offset, width
```

where:

- `char` is the font character (e.g. `f`, `@`, `!`)
- `offset` is the font character's distance in pixels from the left of the image
- `width` is its width in pixels

7.3.17 [Floors]

The `Floors` section contains details of the "floors" in the skool. A "floor" (note the quotes) is a region of the skool that cannot be reached from another region of the skool without navigating a staircase. For example, in Skool Daze, the region to the left of the Map Room wall is one floor, and the region to the right of the Map Room wall is another floor. You can't get from one to other without going up or down a staircase (walking through walls is prohibited).

Each line in this section has the form:

```
floorId, minX, maxX, y
```

where:

- `floorId` is the floor's unique ID
- `minX` and `maxX` are the x-coordinates of the left and right limits of the floor
- `y` is the y-coordinate of the floor (3 = top floor, 10 = middle floor, 17 = bottom floor)

The unique IDs are used in the *[Routes]* section.

7.3.18 [Frogs]

Each line in the `Frogs` section has the form:

```
frogId, spriteGroupId, animatoryState, (x, y), commandListId, turnRound, shortHop, longHop, sitXY, e
```

where:

- `frogId` is the unique ID of the frog
- `spriteGroupId` is the ID of the *sprite group* to use for the frog
- `animatoryState` is the frog's initial animatory state
- `(x, y)` are the frog's initial coordinates
- `commandListId` is the unique ID of the *command list* that the frog will use
- `turnRound` is the ID of the sequence of *animation phases* to use when the frog turns round
- `shortHop` is the ID of the sequence of *animation phases* to use when the frog makes a short hop
- `longHop` is the ID of the sequence of *animation phases* to use when the frog makes a long hop
- `sitXY` is the coordinates of the frog within its sprite when it's sitting (used for collision detection and placement in cups)

- `ericProximity` is the minimum distance from the frog that Eric can be before it will try to hop away

Any frog defined in this section will be catchable by ERIC, and show up in the on-screen inventory when caught.

7.3.19 [GameConfig]

The `GameConfig` section contains configuration parameters in the format:

```
parameterName, parameterValue
```

Recognised parameters are:

- `AllShieldsScore` - points awarded for hitting all the shields
- `AssemblyHallId` - ID of the assembly hall (as defined in the *[Rooms]* section); this is used to check whether Eric can sit or should be sitting on the floor
- `AssemblySitDirection` - the direction Eric should face when sitting down for assembly (`-1` for left, `1` for right)
- `BesideEricXRange` - maximum horizontal distance from Eric at which a character can be to be considered beside him
- `BikeCombinationScore` - points awarded for writing the bike combination on a blackboard
- `BikeSecrets` - valid bike combination characters
- `Cheat` - 0 = disable cheat keys, 1 = enable cheat keys
- `ConkerClockTicks` - the number of ticks by which the skool clock is rewound (that is, the number of ticks by which the current period is extended) when a character is paralysed by a falling conker
- `ConkerWakeTime` - the time (clock ticks remaining before the next bell ring) at which a character paralysed by a conker will remobilise
- `DrinksCabinetDoorId` - the ID of the drinks cabinet door (see *[Doors]*); this is used to detect whether Eric has jumped up to it (to get the sherry)
- `EvadeMouseDelay` - the delay before a character frightened by a mouse will either get off a chair or stop jumping
- `ExpellerId` - the ID of the character who is responsible for expelling Eric
- `FireCatapultProbability` - the probability that the tearaway will fire his catapult if conditions are suitable
- `HitProbability` - the probability that the bully will throw a punch if conditions are suitable
- `Icon` - the name of the icon file to use
- `ImageSet` - the name of the image set to use
- `GameFps` - the number of frames per second at which the game should attempt to run; raise it to increase the game speed, or lower it to decrease the game speed
- `KissCounter` - the initial value of the kiss counter for a character
- `KissCounterDeckrement` - the amount by which a character's kiss counter is decreased after being knocked over
- `KissCounterDecrement` - the amount by which a character's kiss counter is decreased after kissing Eric
- `KissDistance` - the exact distance in front of Eric a character must be in order to be kissable
- `KissLines` - the number of lines a kissee does for Eric on each kiss

- `LinesGivingRange` - the maximum horizontal and vertical distances a character must be away from a target character to be considered close enough to give or be given lines
- `LinesRange` - minimum and maximum number of lines (divided by 100) that may be given out in one go
- `LocationMarker` - prefix used in a destination ID to denote the location of a character
- `MaxLines` - the maximum number of lines Eric may accumulate before being expelled
- `MaxMiceRelease` - the maximum number of mice to release per attempt
- `MinimumLinesDelay` - the minimum delay between two non-immediate lines-givings by the same teacher
- `MouseCatchScore` - points awarded for catching a mouse
- `MouseProximity` - maximum distance at which a musophobe can detect a mouse (and so be scared by it)
- `Name` - the name of the game
- `Playground` - the x-coordinates of the left and right boundaries of the playground (used for checking whether Eric's in the playground)
- `PlayTuneOnRestart` - 1 to play the theme tune after restarting the game for advancing a year; 0 otherwise
- `QuickStart` - 0 to scroll the skool into view and play the theme tune (as in the original games); 1 to skip this sequence
- `RestartOnYearEnd` - 1 if the game should restart after advancing a year (as in Back to Skool); 0 otherwise (as in Skool Daze)
- `SafeKeyScore` - points awarded when the safe key is obtained
- `SafeOpenScore` - points awarded for opening the safe with the correct combination
- `SafeSecrets` - valid safe combination characters
- `SaveGameDir` - the directory in which saved games will be stored
- `SaveGameCompression` - the compression level to use when saving a game (0 = no compression, 9 = maximum compression)
- `ScreenshotDir` - the directory in which screenshots are dumped
- `SherryId` - the ID to use for sherry fired from a water pistol; by default this is different from the value of `WaterId` so that sherry will not make plants grow
- `SpriteSize` - the width and height of a sprite (in tiles)
- `StoreroomCombinationScore` - points awarded for writing the storeroom combination on a blackboard
- `StoreroomDoorId` - the ID of the science lab storeroom door (see *[Doors]*); this is used to detect whether Eric can open a door with the storeroom key
- `StoreroomSecrets` - valid storeroom combination characters
- `TooManyLinesCommandList` - the ID of the command list Mr Wacker should use to expel Eric after he's exceeded the lines limit
- `UpAYearScore` - points awarded for advancing a year
- `WaterId` - the ID to use for water fired from a water pistol; liquid with this ID will make plants grow (see `SherryId`)
- `WindowProximity` - maximum distance at which a window is considered nearby (i.e. worth opening if a stinkbomb is smelt)

7.3.20 [GrassMessages]

The `GrassMessages` section contains five lines of the form:

```
Writers, characterId[, characterId...]
WriteTale, <text>
Hitters, characterId[, characterId...]
HitTale, <text>
AbsentTale, <text>
```

The information in this section is used by the swot to determine who can be blamed for hitting him or writing on a blackboard, and what to say when telling tales.

The `Writers` line contains a comma-separated list of IDs of characters who can be blamed for writing on a blackboard. The `WriteTale` line contains the text of the blackboard-writing tale.

The `Hitters` line contains a comma-separated list of IDs of characters who can be blamed for hitting the swot. The `HitTale` line contains the text of the so-and-so-hit-me tale.

The `AbsentTale` line contains the text that will be spoken by the swot when he's telling on Eric for being absent during class.

The text of a tale may contain any of the following macros:

- `$TITLE` - which will be replaced by the teacher's title, as defined in the *[Characters]* section; to change this macro, set the `TitleMacro` configuration parameter in the *[MessageConfig]* section
- `$1` - which will be replaced by the name of the hitter or writer chosen from the `Hitters` or `Writers` list; to change this macro, set the `GrasseeMacro` configuration parameter in the *[MessageConfig]* section
- `$characterId` (where `characterId` is the unique ID of any character) - which will be replaced by the name of that character

7.3.21 [Images]

Each line in the `Images` section has the form:

```
imageId, path
```

where

- `imageId` is the unique ID of an image
- `path` is the location of the corresponding image file on disk (relative to the *images* directory)

Recognised image IDs and the images they refer to are:

- `FONT`: the skool font
- `INVENTORY`: mouse, frog, water pistol etc. (Back to Skool only)
- `LESSON_BOX`: the lesson box background
- `LOGO`: the logo
- `MESSAGE_BOX`: the message box used to display messages above a character's head (lines messages, escape alarm messages, and safe, bike, and storeroom combination characters)
- `MUTABLES`: doors, windows etc. (Back to Skool) or shields and safe (Skool Daze) - full colour
- `MUTABLES_INK`: doors, windows etc. (Back to Skool) or shields and safe (Skool Daze) - ink only
- `MUTABLES_PAPER`: doors, windows etc. (Back to Skool) or shields and safe (Skool Daze) - paper only

- SCOREBOX: , the score/lines/hi-score box background
- SKOOL: the skool - full colour
- SKOOL_INK: the skool - ink colours only
- SKOOL_PAPER: the skool - paper colours only
- SPEECH_BUBBLE: speech bubble and lip
- SPRITES: the characters in various 'animatory states'

7.3.22 [Inventory]

Each line in the Inventory section has the form:

```
itemId, topLeft, size
```

where:

- itemId is the unique ID of an item that can be collected
- topLeft is the coordinates (in *(x, y)* form) of the top left of the image of the item in *inventory.png*
- size is the size of the image (in *(width, height)* form)

The item IDs recognised by Pyskool are as follows:

- FROG - a frog
- MOUSE - a mouse
- SAFE_KEY - the key to the head's safe
- SHERRY_PISTOL - a water pistol (containing sherry)
- STINKBOMBS3 - three stinkbombs
- STINKBOMBS2 - two stinkbombs
- STINKBOMBS1 - one stinkbomb
- STOREROOM_KEY - the key to the science lab storeroom
- WATER_PISTOL - a water pistol (containing water)

The order in which the items appear in the Inventory section determines the order in which they will be printed in the on-screen inventory.

See also the *[Mice]* and *[Frogs]* sections (for details on those animals), and the InventoryPos and MouseInventoryPos configuration parameters in the *[ScreenConfig]* section.

7.3.23 [Lesson ...]

The [Lesson ...] section names take the form:

```
Lesson lessonId [*]characterId, roomId
```

if the lesson will take place with a teacher in a classroom or the dinner hall, or:

```
Lesson lessonId locationId
```

if the lesson is an unsupervised period, where:

- lessonId is the lesson ID as it appears in the *[Timetable]* section

- `characterId` is the character ID of the teacher taking Eric's class (prefixed by '*' if the teacher's name should not be printed in the lesson box, as during `DINNER`)
- `roomId` is the ID of the room in which Eric's class will take place
- `locationId` is one of `PLAYTIME`, `REVISION LIBRARY`, and `ASSEMBLY`

Each line in a `[Lesson ...]` section has the form:

```
characterId, commandListId
```

where

- `characterId` is the unique ID of a character (see *[Characters]*)
- `commandListId` is the ID of the *command list* that will control the character's movements during the lesson

A command list is a sequence of commands - such as *GoTo* or *FindSeat* - that make a character appear intelligent (kind of). See *[CommandList ...]* for more details.

In any `[Lesson ...]` section there should be one line for each character defined in the *[Characters]* section.

7.3.24 [LessonConfig]

The `LessonConfig` section contains configuration parameters in the format:

```
parameterName, parameterValue
```

Recognised parameters are:

- `BlackboardBacktrack` - the distance a teacher walks back after wiping a blackboard
- `BlackboardPaceDistance` - the distance a teacher should pace up and down in front of the blackboard during a lesson without a question-and-answer session
- `EricsTeacherWriteOnBoardProbability` - the probability that a teacher will write on the blackboard during a lesson with Eric and the swot
- `GrassForHittingProbability` - the probability that the swot will grass on someone for hitting him
- `LinesForTalesProbability` - the probability that the teacher will give the swot lines for telling tales
- `QASessionProbability` - the probability that the teacher will start a question-and-answer session with the swot
- `WriteOnBoardProbability` - the probability that a teacher will write on the blackboard during a lesson without Eric and the swot

7.3.25 [LessonMessages]

The `LessonMessages` section contains a list of messages that will be used by teachers who are not teaching Eric, or teachers who are teaching Eric but have chosen not to do a question-and-answer session. Each line in the section takes the form:

```
characterId|*, lessonMessage[, condition]
```

where:

- `characterId` is the unique ID of a teacher
- `lessonMessage` is the message to add to that teacher's repertoire

- `condition` is a condition identifier that must evaluate to true before the message can be used

If `*` is used instead of a specific character ID, the message will be placed in every teacher's repertoire.

A lesson message may contain a character sequence `$(N, M)` (where *N* and *M* are numbers); if so, it will be replaced by a random number between *N* and *M*.

The only recognised condition identifier is:

- `BoardDirty`

(as defined by the `BoardDirtyConditionId` parameter in the *[MessageConfig]* section) which, if specified, means the message will be used only if the blackboard (if there is one) has been written on. Any other condition identifier will evaluate to true.

7.3.26 [LinesMessages]

The `LinesMessages` section contains a list of admonitions delivered by lines-givers when Eric has been spotted doing something he shouldn't. Each line in this section has the form:

```
characterId|*, linesMessageId, linesMessage
```

where

- `characterId` is the unique ID of the lines-giving character
- `linesMessageId` is the unique ID of the following message
- `linesMessage` is the admonition itself

If `*` is used instead of a character ID, the lines message will be used by all lines-givers (unless they have been explicitly assigned a lines message with the same lines message ID). For example:

```
WITHIT, NO_HITTING, BE GENTLE^NOW
*, NO_HITTING, DON'T HIT^YOUR MATES
```

would make Mr Withit scream "BE GENTLE NOW" whenever he sees Eric throwing a punch, whereas every other teacher would scream "DON'T HIT YOUR MATES" instead.

A lines message always spans two lines on-screen. A caret (`^`) is used by default to indicate where the words should be wrapped; to change this, set the `Newline` configuration parameter in the *[MessageConfig]* section.

Lines message IDs used in both Skool Daze and Back to Skool are:

- `BE_PUNCTUAL` - Eric was late for class
- `COME_ALONG_1` - the truant Eric is being guided to the classroom
- `COME_ALONG_2` - the truant Eric is still being guided to the classroom
- `GET_ALONG` - Eric is not in class when he should be
- `GET_OUT` - Eric's somewhere that only staff are allowed to be
- `GET_UP` - Eric is sitting on the floor
- `NEVER_AGAIN` - teacher thinks Eric knocked him down
- `NO_CATAPULTS` - catapults are forbidden
- `NO_HITTING` - Eric must not throw punches
- `NO_TALES` - the swot gets his just deserts
- `NO_WRITING` - Eric mustn't write on blackboards

- SIT_DOWN - Eric is not sitting down in class
- STAY_IN_CLASS - Eric popped out of class and then returned

Lines message IDs used only in Skool Daze are:

- COME_ALONG_3 - the truant Eric still hasn't made it to the classroom
- NO_JUMPING - jumping is not allowed in Skool Daze
- NO_SITTING_ON_STAIRS - sitting on stairs is not allowed in Skool Daze

Lines message IDs used only in Back to Skool are:

- BACK_TO_SKOOL - Eric should be back in the boys' skool by now
- GET_OFF_PLANT - standing on plants is not good
- NO_BIKES - riding bikes inside the skool is also not good
- NO_STINKBOMBS - stinkbombs are prohibited
- NO_WATERPISTOLS - as are water pistols
- SIT_FACING_STAGE - Eric needs to face the headmaster during assembly

The lines message IDs are used internally, and should not be changed. If a particular lines message ID is missing from the list, then lines will not be given for the infraction it refers to. So if there were no entry in the LinesMessages section with the lines message ID NO_HITTING, no lines would ever be dished out for hitting.

7.3.27 [MessageConfig]

The MessageConfig section contains messages and message-related configuration parameters that apply skool-wide. Each line in this section has the form:

```
parameterName, parameterValue
```

Recognised parameters are:

- BoardDirtyConditionId - the ID of the condition used to indicate that a blackboard is dirty; this identifier may be used in the *[LessonMessages]* section
- GrasseeMacro - the macro that expands to a grassee's name in the swot's speech
- HiScoreLabel - the label for the hi-score in the score box
- LinesMessageTemplate - the template used for lines messages
- LinesRecipientMacro - the macro that will be replaced in LinesMessageTemplate (see above) by the lines recipient's name
- LinesTotalLabel - the label for the lines total in the score box
- Newline - the character that will be replaced by a newline character in messages written on a blackboard, in a lines bubble, or in the lesson box
- NumberOfLinesMacro - the macro that will be replaced in LinesMessageTemplate (see above) by the number of lines being given
- ScoreLabel - the label for the score in the score box
- TitleMacro - the macro that expands to a teacher's title in the swot's speech
- UpAYearMessage - the message printed in the lesson box when Eric has completed the game and advanced a year

7.3.28 [Mice]

Each line in the `Mice` section has the form:

```
mouseId, spriteGroupId, animatoryState, (x, y), commandListId, spriteXY
```

where:

- `mouseId` is the unique ID of the mouse
- `spriteGroupId` is the ID of the *sprite group* to use for the mouse
- `animatoryState` is the mouse's initial animatory state
- `(x, y)` are the mouse's initial coordinates
- `commandListId` is the ID of the *command list* that the mouse will use
- `spriteXY` is the coordinates of the mouse within its sprite (used for detecting whether Eric has caught it)

Any mouse defined in this section will be catchable by ERIC, and show up in the on-screen mouse inventory when caught.

7.3.29 [MouseLocations]

The `MouseLocations` section defines the locations at which a new immortal mouse may appear after Eric catches one. Each line has the form:

```
x, y
```

where `(x, y)` are the coordinates of the location.

7.3.30 [NoGoZones]

Each line in the `NoGoZones` section corresponds to a region of the skool Eric is never supposed to enter. The lines take the form:

```
zoneId, minX, maxX, bottomY, topY
```

where:

- `zoneId` is a descriptive ID for the zone (not used anywhere else)
- `minX` is the lower x-coordinate of the zone
- `maxX` is the upper x-coordinate of the zone
- `bottomY` is the y-coordinate of the bottom of the zone
- `topY` is the y-coordinate of the top of the zone

Whenever Eric is spotted in one of these zones by a teacher, the `GET_OUT` *lines message* will be delivered in screeching tones.

7.3.31 [Plants]

The `Plants` section contains information about plants (of which there are four in Back to Skool, and none in Skool Daze). Each line describes a single plant, and has the form:

```
plantId, spriteGroupId, x, y, commandListId
```

where:

- `plantId` is the unique ID of the plant
- `spriteGroupId` is the ID of the *sprite group* to use for the plant
- `x` and `y` are the coordinates of the plant (when it is growing or has grown)
- `commandListId` is the unique ID of the *command list* that the plant will use when watered

7.3.32 [QuestionsAndAnswers ...]

The `[QuestionsAndAnswers ...]` section names take the form:

```
QuestionsAndAnswers characterId
```

where `characterId` is the unique ID of a teacher (see *[Characters]*).

There are at least three types of entry in a `[QuestionsAndAnswers ...]` section. The first type of entry is the `Question` entry:

```
Question, questionId, groupId, questionTemplate
```

where:

- `questionId` is a unique (within the section) ID for the question
- `groupId` is the ID of the group of Q&A pairs (see below) the question is associated with
- `questionTemplate` is the question template

There should be at least one `Question` entry in a `[QuestionsAndAnswers ...]` section.

The second type of entry is the `Answer` entry:

```
Answer, questionId, answerTemplate
```

where:

- `questionId` is the ID of the question to which this is the answer
- `answerTemplate` is the answer template

There should be one `Answer` entry for each `Question` entry in a `[QuestionsAndAnswers ...]` section.

The third type of entry in this section is the Q&A pair entry:

```
groupId, word1, word2
```

where

- `groupId` is the ID of the group of Q&A pairs to which this particular pair belongs; the ID should be something other than *Question*, *Answer*, *SpecialGroup*, *SpecialQuestion* or *SpecialAnswer*, which are reserved words in a `[QuestionsAndAnswers ...]` section
- `word1` and `word2` are the words that will replace the macros in `questionTemplate` and `answerTemplate`

There should be at least one Q&A pair defined per `[QuestionsAndAnswers ...]` section (and ideally many more than one, to prevent the question-and-answer sessions between teachers and the swot from being rather monotonous).

The optional fourth type of entry in a `[QuestionsAndAnswers ...]` section consists of three lines:

```
SpecialQuestion, question
SpecialAnswer, answer
SpecialGroup, groupId, qaPairIndex
```

and is used to define the "special" question Eric will need the answer to in order to obtain the relevant teacher's safe combination letter. The `SpecialQuestion` keyword is followed by the text of the special question (which will be posed by the teacher at the start of the lesson). The `SpecialAnswer` keyword is followed by the text of the swot's answer to the special question (which will contain a macro to be expanded). The `SpecialGroup` keyword is followed by `groupId` (which specifies the ID of the group of Q&A Pairs from which the "magic word" will be taken), and `qaPairIndex` (which is 0 or 1, and refers to the element of the Q&A pair that will be the magic word). Once Eric has figured out what the magic word is, he will need to write it on a blackboard and hope that the teacher sees it while all the shields are flashing, whereupon the teacher will reveal his safe combination letter.

If the `SpecialQuestion`, `SpecialAnswer` and `SpecialGroup` lines are not present, there will be no magic word associated with the teacher. In that case, simply knocking the teacher over with a catapult pellet will make him reveal his safe combination letter.

7.3.33 [RandomLocations]

The `RandomLocations` section contains lists of suitable locations for the characters to visit when they go on walkabouts (e.g. during playtime). Each line has the form:

```
characterId, (x1, y1), (x2, y2)...
```

where:

- `characterId` is the character's unique ID (see *[Characters]*)
- `(x1, y1)` and so on are the coordinates of locations in the skool

There must be at least one pair of coordinates per line, and there should be one line for each character defined in the *[Characters]* section.

7.3.34 [Rooms]

The `Rooms` section contains one line for each room or region in the skool that Eric will be expected to show up at when the timetable requires it. Each line has the form:

```
roomId, name, topLeft, bottomRight, getAlong
```

where:

- `roomId` is the room's unique ID
- `name` is the room's name (as displayed in the lesson box at the bottom of the screen)
- `topLeft` is the coordinates of the top-left corner of the room
- `bottomRight` is the coordinates of the bottom-right corner of the room
- `getAlong` is `Y` if Eric should get lines for being in the room when the timetable does not require his presence

7.3.35 [Routes]

The `Routes` section is one of the most important sections in the ini file. It defines the routes (a route may be considered as a list of staircases) that the characters must take to get from where they are to wherever they are going.

Any errors here will result in the characters wandering aimlessly around the skool, unable to find classrooms, the playground, or the toilets. Disaster!

Anyway, each line in this section has the form:

```
homeFloorId, *|destFloorId[, destFloorId[, ...]], nextStaircaseId
```

where:

- `homeFloorId` is the unique ID of one floor (see *[Floors]*) - the 'home' floor
- `destFloorId` is the unique ID of another floor (see *[Floors]*) - the destination floor
- `nextStaircaseId` is the unique ID of the staircase (see *[Staircases]*) that must be climbed or descended first on a journey from the home floor to the destination floor

How this works is best illustrated by example. Let's look at the routes defined for the bottom floor in Back to Skool to everywhere else in the skool:

```
Bottom, LeftMiddle, LeftTop, UpToToilets
Bottom, GirlsMiddle, GirlsTop, GirlsSkoolLower
Bottom, *, UpToStage
```

The first line says that to get from the bottom floor (`Bottom`) to the floors called `LeftMiddle` and `LeftTop` (see *[Floors]*), the first staircase you need to navigate is `UpToToilets` (see *[Staircases]*). The second line says that to get from the bottom floor to the middle floor (`GirlsMiddle`) or top floor (`GirlsTop`) in the girls' skool, you need to use the `GirlsSkoolLower` staircase first. The third line says that to get anywhere else (`*`) from the bottom floor, you need to take the stairs up to the stage (`UpToStage`).

7.3.36 [Safe]

The `Safe` section contains a single line of the form:

```
topLeft, size, coords
```

where:

- `topLeft` is the coordinates (in *(x, y)* form) of the top left of the normal image of the safe
- `size` is the size of the image (in *(width, height)* form)
- `coords` are the coordinates (in *(x, y)* form) of the safe in the play area

The safe images can be found in *mutables.png* (or *mutables_ink.png* and *mutables_paper.png* if `GraphicsMode` is 1 - see *[ScreenConfig]*). *mutables.png* is arranged so that the inverse image of the safe is at *(x + width, y)*, where *(x, y)* are the coordinates of the normal image of the safe.

If the safe will never need to flash (as in Back to Skool), `topLeft` and `size` will not be used, and so may be set to any value.

7.3.37 [ScreenConfig]

The `ScreenConfig` section contains parameters that determine the appearance and layout of the screen. Each line has the form:

```
parameterName, parameterValue
```

Recognised parameters are:

- `Background` - the background colour of the screen

- `EscapeAlarmInk` - the ink colour to use for the escape alarm message box used by Albert
- `EscapeAlarmPaper` - the paper colour to use for the escape alarm message box used by Albert
- `FlashCycle` - length of the cycle in which a flashable object (such as a shield) flashes once
- `FontInk` - the ink colour in *font.png* (used to create transparency)
- `FontPaper` - the paper colour in *font.png* (used to create transparency)
- `GraphicsMode` - 0 = hi-res colour; 1 = spectrum mode, meaning just two colours (ink and paper) per 8x8-pixel block
- `Height` - the height of the screen (in tiles)
- `HiScoreOffset` - the y-coordinate offset used to position the printing of the hi-score
- `InitialColumn` - the x-coordinate of the leftmost column of the screen when the game starts
- `InventoryKey` - pixels of this colour in inventory item and captured mouse images will be made transparent when the items are drawn
- `InventoryPos` - the x, y coordinates of the inventory on screen
- `InventorySize` - the size of the inventory (width and height in tiles)
- `LessonBoxInk` - the ink colour to use when writing in the lesson box
- `LessonBoxPos` - the x, y coordinates of the lesson box on screen
- `LinesInk` - the ink colour used in a lines message box
- `LinesOffset` - the y-coordinate offset used to position the printing of the lines total
- `LinesPaperEric` - the paper colour used in a lines message box when Eric is the recipient
- `LinesPaperOther` - the paper colour used in a lines message box when Eric is not the recipient
- `LogoPos` - the x, y coordinates of the logo on screen
- `MessageBoxColour` - the colour of the 'inside' of the message box in the `MESSAGE_BOX` image (see *[Images]*); pixels of this colour in the image will take on the designated paper colour (e.g. `LinesPaperEric`) when the message box is drawn
- `MessageBoxKey` - pixels of this colour in the message box image will be made transparent when the message box is drawn; in the stock Pyskool, this feature is not used
- `MouseInventoryInk` - the ink colour to use when writing in the mouse inventory
- `MouseInventoryPos` - the x, y coordinates of the mouse inventory on screen
- `MouseInventorySize` - the size of the mouse inventory (width and height in tiles)
- `Scale` - the scale factor to use for graphics; 1 = original Spectrum size
- `ScoreBoxInk` - the ink colour to use when writing in the score box
- `ScoreBoxPos` - the x, y coordinates of the score box on screen
- `ScoreOffset` - the y-coordinate offset used to position the printing of the score
- `ScrollFps` - the number of frames per second at which the screen should be scrolled (when the game starts and during play); raise it to make the screen scroll faster, or lower it to scroll more slowly
- `ScrollColumns` - the number of columns to scroll when Eric approaches the left or right edge of the screen
- `ScrollLeftOffset` - how close Eric can get to the right edge of the screen before it scrolls left
- `ScrollRightOffset` - how close Eric can get to the left edge of the screen before it scrolls right

- `SecretInk` - the ink colour of the message box used to display a safe, bike or storeroom combination character
- `SecretPaper` - the paper colour of the message box used to display a safe, bike or storeroom combination character
- `SkoolInkKey` - the transparent colour used in the skool ink image
- `SpeechBubbleInk` - the ink colour to use when drawing text in a speech bubble
- `SpeechBubbleKey` - the transparent colour used in the speech bubble image (*bubble.png*)
- `SpeechBubbleInset` - the inset (in pixels at scale 1) of the text window from the top-left of a speech bubble
- `SpeechBubbleLipCoords` - the coordinates of the lip within the speech bubble image (*bubble.png*)
- `SpeechBubbleLipSize` - the size of the speech bubble lip (width and height in tiles)
- `SpeechBubbleSize` - the size of the bounding rectangle of a speech bubble, including the lip (width and height in tiles)
- `SpriteKey` - the transparent colour used in the sprite matrix image
- `SpriteMatrixWidth` - the number of sprites in a row of the sprite matrix image
- `Width` - the width of the screen (in tiles)

7.3.38 [SherryDrop]

The `SherryDrop` section defines a drop of sherry (as knocked out of a cup by a catapult pellet). It contains a single line of the form:

```
objectId, spriteGroupId, commandListId, hitXY
```

where:

- `objectId` is a unique ID for the drop of sherry
- `spriteGroupId` is the ID of the *sprite group* to use for the drop of sherry
- `commandListId` is the unique ID of the *command list* that the drop of sherry will use when knocked out of a cup
- `hitXY` is the coordinates of the sherry drop within its sprite (used for collision detection)

7.3.39 [Shields]

The `Shields` section contains information about shields (of which there are 15 in Skool Daze, and none in Back to Skool). Each line describes a single shield, and has the form:

```
score, topLeft, size, coords
```

where:

- `score` is the number of points awarded for making the shield flash or unflash
- `topLeft` is the coordinates (in *(x, y)* form) of the top left of the normal image of the shield
- `size` is the size of the image (in *(width, height)* form)
- `coords` are the coordinates (in *(x, y)* form) of the shield in the play area

The shield images can be found in *mutables.png* (or *mutables_ink.png* and *mutables_paper.png* if `GraphicsMode` is 1 - see *[ScreenConfig]*). *mutables.png* is arranged so that the inverse image of a shield is at *(x + width, y)*, where *(x, y)* are the coordinates of the normal image of the shield.

7.3.40 [SitDownMessages]

The `SitDownMessages` section contains one or more lines for each teacher of the form:

```
characterId, sitDownMessage
```

where

- `characterId` is the teacher's unique ID (see *[Characters]*)
- `sitDownMessage` is what the teacher may say while standing at the classroom doorway at the start of a lesson

If multiple sit-down messages are defined for a teacher, he will choose one at random when the time comes. If no sit-down messages are defined for a teacher, he will say nothing at the classroom doorway.

7.3.41 [SkoolLocations]

The `SkoolLocations` section contains a list of descriptive IDs for commonly used locations in the skool. These descriptive IDs are used by the *GoTo* command in the *command lists* that control the characters. Each line in this section has the form:

```
locationId, x, y
```

where

- `locationId` is the descriptive ID
- `x` and `y` are the coordinates of the location

An example of a location ID is `BlueRoomDoorway`, which means exactly what you think it means.

7.3.42 [Sounds]

Each line in the `Sounds` section has the form:

```
soundId, path
```

where

- `soundId` is the unique ID of a sound effect
- `path` is the location of the sound file on disk (relative to the *sounds* directory)

`path` may be the full name of the sound file (e.g. *tune.wav*), or just the base name (e.g. *tune*); in the latter case, Pyskool will look for a file with the base name and a *.wav* or *.ogg* suffix.

Recognised IDs and the sound effects they refer to are:

- `ALARM`: Albert is telling Mr Wacker that Eric is escaping
- `ALL_SHIELDS`: Eric has hit all the shields
- `BELL`: the bell
- `BIKE`: Eric has written the bike combination on a blackboard
- `CATAPULT`: Eric has fired his catapult
- `CONKER`: Eric has knocked out Albert with a conker
- `DESK`: Eric has found the water pistol or stinkbombs in a desk
- `FROG`: Eric has caught the frog or placed it in a cup

- HIT0, HIT1: Eric has thrown a punch
- JUMP: Eric has jumped into the air
- KISS: Eric has kissed someone
- KNOCKED_OUT: Eric has been knocked over or out of his chair
- LINES1: lines screech 1
- LINES2: lines screech 2
- MOUSE: Eric has caught a mouse
- OPEN_SAFE: Eric has opened the safe (by getting the combination)
- SAFE_KEY: Eric has got the safe key
- SHERRY: Eric has filled the water pistol with sherry
- SHIELD: Eric has hit a shield
- STOREROOM_KEY: Eric has written the storeroom combination on a blackboard
- TUNE: opening tune
- UP_A_YEAR: Eric has gone up a year
- WALK0, WALK1, WALK2, WALK3: Eric walking
- WATER_PISTOL: Eric has fired his water pistol

If an entry for a given sound effect is not present in the Sounds section, then that sound effect will never play. For example, if there is no CATAPULT entry, then Eric's firings of that weapon will be completely silent.

7.3.43 [SpecialPlaytimes]

The SpecialPlaytimes section is used only by Skool Daze, and contains a list of lesson IDs that refer to playtimes which will be considered "special". A special playtime does not appear in the timetable proper (though you could insert it), but with a given probability (defined by the SpecialPlaytimeProbability parameter in the *[TimetableConfig]* section) a special playtime chosen at random is substituted for an actual playtime from the main timetable. In Skool Daze the SpecialPlaytimes section looks like this:

```
Playtime-Mumps
Playtime-SwotGrass
Playtime-HiddenPeaShooter
```

Thus, occasionally in Skool Daze mode a playtime will be one of those where you have to steer clear of the pestilential bully, prevent the swot from reaching the head's study, or fix the race to the fire escape between the tearaway and the headmaster.

7.3.44 [SpriteGroup ...]

The [SpriteGroup ...] section names take the form:

```
SpriteGroup spriteGroupId
```

where spriteGroupId is a unique ID for a group of sprites in *sprites.png* (see *Graphics*) - such as BOY for the little boys, or TEARAWAY for the tearaway. The unique ID can be anything you like; it is used only in the *[Characters]* section later on to link a character to a specific group of sprites.

Each line in a SpriteGroup section represents a single sprite from *sprites.png* and has the form:

```
spriteId, index
```

where

- `spriteId` is the descriptive ID for the sprite (unique within the section)
- `index` is the index of the sprite as it appears in *sprites.png*

Recognised sprite IDs and their meanings are:

- `ARM_UP`: arm up (as if writing or opening door) - Eric, the tearaway, the Heroine and teachers
- `BENDING_OVER`: bending over - Eric
- `BIKE_ON_FLOOR`: bike resting on the floor
- `BIKE_UPRIGHT`: bike upright
- `CATAPULT0`: firing catapult (1) - Eric and the tearaway
- `CATAPULT1`: firing catapult (2) - Eric and the tearaway
- `CONKER`: conker
- `DESK_EMPTY`: desk lid (empty desk)
- `DESK_STINKBOMBS`: desk lid (with stinkbombs)
- `DESK_WATER_PISTOL`: desk lid (with water pistol)
- `FLY`: catapult pellet in flight
- `HITTING0`: hitting (1) - Eric and the bully
- `HITTING1`: hitting (2) - Eric and the bully
- `HOP1`: frog hopping (phase 1)
- `HOP2`: frog hopping (phase 2)
- `KISSING_ERIC`: kissing Eric - the Heroine
- `KNOCKED_OUT`: lying flat on back - kids
- `KNOCKED_OVER`: sitting on floor holding head - adults
- `PLANT_GROWING`: plant (half-grown)
- `PLANT_GROWN`: plant (fully grown)
- `RIDING_BIKE0`: riding bike (1) - Eric
- `RIDING_BIKE1`: riding bike (2) - Eric
- `RUN`: mouse
- `SHERRY_DROP`: drop of sherry (knocked from a cup)
- `SIT`: frog sitting
- `SITTING_ON_CHAIR`: sitting on a chair - kids
- `SITTING_ON_FLOOR`: sitting on the floor - kids
- `STINKBOMB`: stinkbomb cloud
- `WALK0`: standing/walking (1) - all characters
- `WALK1`: midstride (1) - all characters
- `WALK2`: standing/walking (2) - all characters

- `WALK3`: midstride (2) - all characters
- `WATER_DROP`: drop of water (knocked from a cup)
- `WATER0`: water fired from a pistol (phase 1)
- `WATER1`: water fired from a pistol (phase 2)
- `WATER2`: water fired from a pistol (phase 3)
- `WATER3`: water fired from a pistol (phase 4)
- `WATER4`: water fired from a pistol (phase 5)
- `WATERPISTOL`: shooting water pistol - Eric

7.3.45 [Staircases]

The `Staircases` section contains details of the staircases in the skool. Each line has the form:

```
staircaseId[:alias], bottom, top[, force]
```

where:

- `staircaseId` is the staircase's unique ID
- `alias` is an optional alias for the staircase (also unique)
- `bottom` and `top` are the coordinates of the bottom and top of the staircase (in *(x, y)* form)
- `force`, if present, indicates that the staircase must be climbed or descended by Eric if he moves to a location between the bottom and the top

In the stock Pyskool, `force` is used only for the staircase in Back to Skool that leads down to the assembly hall stage; it's the only staircase that you must go up or down if you approach it.

An example of a line from the `Staircases` section is:

```
UpToStudy:DownFromStudy, (91, 10), (84, 3)
```

which defines the staircase that leads up to the head's study in Back to Skool. This staircase's unique ID is `UpToStudy`, but it can also be referred to as `DownFromStudy`. These unique IDs and aliases are used in the *[Routes]* section.

7.3.46 [Stinkbombs]

Each line in the `Stinkbombs` section has the form:

```
characterId, stinkbombId, spriteGroupId, commandListId, animationPhases, stinkRange
```

where:

- `characterId` is the unique ID of the character to give stinkbomb-dropping ability to
- `stinkbombId` is the unique ID of the stinkbomb
- `spriteGroupId` is the ID of the *sprite group* to use for the stinkbomb when dropped
- `commandListId` is the unique ID of the *command list* that the stinkbomb will use when dropped
- `animationPhases` is the ID of the sequence of *animation phases* that the stinkbomb cloud will use
- `stinkRange` - the maximum distance at which the stinkbomb can be smelt

Each character whose unique ID appears in this section will be given the ability to drop a stinkbomb. In the stock Pyskool this will be Eric.

7.3.47 [Timetable]

The `Timetable` section contains an ordered list of lesson IDs. Lessons happen starting with the first in the list, and proceed one by one to the end of the list. When the last lesson in the list is finished, the game loops back round to the first lesson in the list.

An example of a lesson ID is `Creak-BlueRoom-1`, which refers to the first of a set of lessons in which Eric and the swot are taught by Mr Creak in the Blue Room. The lesson ID could be anything, but it's helpful to make it descriptive.

A lesson can be thought of as a set of entries from the personal timetables of the characters. These sets of entries can be found in the *[Lesson ...]* sections.

7.3.48 [TimetableConfig]

The `TimetableConfig` section contains configuration parameters in the format:

```
parameterName, parameterValue
```

Recognised parameters are:

- `AssemblyPrefix` - what a *lesson ID* must start with to be regarded as Assembly
- `GetAlongTime` - maximum time allowed to leave a classroom or the playground after the bell rings
- `LessonLength` - the length of a lesson period in frames (see `GameFps`)
- `LessonStartTime` - when a lesson starts (i.e. teacher will tell kids to sit down) in frames (see `GameFps`) from the start of the period
- `PlaytimePrefix` - what a *lesson ID* must start with to be regarded as Playtime
- `SpecialPlaytimeProbability` - the probability that a playtime in the main timetable will be replaced by a *special playtime*

7.3.49 [TimingConfig]

The `TimingConfig` section contains configuration parameters in the format:

```
parameterName, parameterValue
```

Recognised parameters are:

- `BendOverDelay` - the delay (in frames) before Eric stands upright after bending over (as when releasing mice)
- `DethronedDelay` - the delay before a character rises after being pushed out of a seat
- `EricWalkDelay` - the number of frames between successive movements of Eric when he's walking
- `JumpDelay` - the delay (in frames) before Eric returns to the floor after jumping
- `KnockedOverDelay` - the delay before a knocked over teacher rises
- `KnockoutDelay` - the delay before a knocked out kid rises
- `GoFast` - the number of frames between successive movements of a character who is moving quickly; this parameter is used when a character is running or speaking

- `GoFaster` - the number of frames between successive movements of a character who is moving even quicker; this parameter is used when a character is throwing a punch or firing a catapult
- `GoSlow` - the number of frames between consecutive movements of a character who is moving slowly; this parameter is used when a character is walking at a normal pace
- `ReprimandDelay` - the delay before a knocked over teacher gives lines to someone for knocking him over
- `SpeedChangeDelayRange` - the minimum and maximum values of the delay between a character's walking speed changes (used by kids, who walk half the time and run the other half)
- `TellEricDelay` - the length of time a character will wait for Eric to respond to a message before repeating it

7.3.50 [Walls]

The `Walls` section contains details of the impenetrable barriers in the skool. Each line has the form:

```
wallId, x, bottomY, topY
```

where:

- `wallId` is the wall's unique ID
- `x` is the wall's x-coordinate
- `bottomY` and `topY` are the y-coordinates of the bottom and top of the wall

For example:

```
FarLeftWall, 0, 20, 0
```

defines the wall at the far left (x=0) of the skool, which stretches from the bottom floor (y=20) to the ceiling of the top floor (y=0).

7.3.51 [Water]

Each line in the `Water` section has the form:

```
characterId, waterId, spriteGroupId, commandListId, animationPhases
```

where:

- `characterId` is the unique ID of the character to give water pistol-firing ability to
- `waterId` is the unique ID for the water sprite
- `spriteGroupId` is the ID of the *sprite group* to use for the water fired from the pistol
- `commandListId` is the unique ID of the *command list* that the water will use when fired from the pistol
- `animationPhases` is the ID of the sequence of *animation phases* that the water will use after being fired from the water pistol

Each character whose unique ID appears in this section will be given the ability to fire a water pistol. In the stock Pyskool this will be Eric alone; he is the only character with a water pistol-firing sprite.

7.3.52 [WaterDrop]

The `WaterDrop` section defines a drop of water (as knocked out of a cup by a catapult pellet). It contains a single line of the form:

```
objectId, spriteGroupId, commandListId, hitXY
```

where:

- `objectId` is a unique ID for the drop of water
- `spriteGroupId` is the ID of the *sprite group* to use for the drop of water
- `commandListId` is the unique ID of the *command list* that the drop of water will use when knocked out of a cup
- `hitXY` is the coordinates of the water drop within its sprite (used for collision detection)

7.3.53 [Windows]

The `Windows` section contains details of the windows in the game. Each line has the form:

```
windowId, x, bottomY, topY, initiallyShut, openerCoords, shutTopLeft, size, coords, descentPhases[, n
```

where:

- `windowId` is the window's unique ID
- `x` is the window's x-coordinate
- `bottomY` and `topY` are the y-coordinates of the bottom and top of the window
- `initiallyShut` is `Y` if the window should be shut when the game starts
- `openerCoords` are the coordinates (in *(x, y)* form) at which a character should stand in order to open the window
- `shutTopLeft` is the coordinates (in *(x, y)* form) of the top left of the image of the window when shut
- `size` is the size of the image (in *(width, height)* form)
- `coords` are the coordinates (in *(x, y)* form) of the window in the skool
- `descentPhases` is the ID of the sequence of *animation phases* to use for Eric if he jumps out of the window
- `notABird` is the ID of the command list Mr Wacker should switch to when Eric hits the ground after falling out of the window; if defined, Eric will be paralysed when he hits the ground

The window images can be found in *mutables.png* (or *mutables_ink.png* and *mutables_paper.png* if `GraphicsMode` is 1 - see *[ScreenConfig]*). *mutables.png* is arranged so that the image of a window when open is at *(x + width, y)*, where *(x, y)* are the coordinates of the image of the same window when shut. The open/shut images for any given window are the same size.

7.4 Commands

The *[CommandList ...]* sections contain commands and parameters that control the characters. Brief descriptions of the commands used in these command lists follow. Unless specified otherwise, commands are used in both Back to Skool and Skool Daze.

7.4.1 AddLines

The `AddLines` command is used to add to Eric's lines total after he has been found guilty of misdeeds. The number of lines to be added to the total is specified by the command's sole argument.

7.4.2 Catch

The `Catch` command is used internally to control Eric while he's trying to catch a mouse or frog. It makes Eric bend over, checks to see whether an animal is present (and if it is, adds it to the appropriate inventory), and then makes Eric stand up.

7.4.3 ChaseEricOut

The `ChaseEricOut` command controls Miss Take as she chases Eric, with the intent of making him leave the girls' skool and return to the boys'. It is very similar to the *ShadowEric* command, except that it makes the headmistress go no further than the skool gate (the x-coordinate of which is supplied as the command's sole argument).

7.4.4 CheckIfTouchingEric

The `CheckIfTouchingEric` command is used by the bully when he has mumps. It checks whether the bully is touching Eric, and if he is, raises the signal that is being watched by whoever is on mumps duty (Mr Rockitt by default).

7.4.5 ConductAssembly

The `ConductAssembly` command makes the headmaster tell the kids they're in detention.

7.4.6 ConductClass

The `ConductClass` command is used by teachers to make them conduct lessons. The command controls a teacher from the point where he reaches the edge of the blackboard. If the teacher is teaching Eric, it makes the teacher wait until the swot shows up, and then hands over control to the *ConductClassWithEric* command. Otherwise it immediately hands over control to the *ConductClassWithoutEric* command.

The command takes two optional arguments:

- the name of the signal that indicates that the swot is ready
- the name of the group of questions and answers to use for the lesson (see *[QuestionsAndAnswers ...]*); if not specified, the questions and answers will be chosen at random

7.4.7 ConductClassWithEric

The `ConductClassWithEric` command is used internally to control a teacher who is teaching Eric and the swot. It takes over control from the *ConductClass* command as soon as the swot shows up and sits down. It's responsible for making the teacher:

- listen to the swot's tales (if any)
- dish out lines after the swot has told tales
- wipe the blackboard
- walk to the middle of the blackboard
- write on the blackboard (occasionally)
- tell the kids what to do during class, or ask questions and wait for the answers
- hunt down Eric if he's playing truant

The command takes a single, optional argument: the name of the group of questions and answers to use for the lesson (see *[QuestionsAndAnswers ...]*); if not specified, the questions and answers will be chosen at random.

7.4.8 ConductClassWithoutEric

The `ConductClassWithoutEric` command is used internally to control a teacher who is teaching a class that does not contain Eric and the swot. It takes over control from the *ConductClass* command immediately. It controls a teacher from the point where he reaches the edge of the blackboard and is responsible for making him:

- wipe the blackboard
- walk to the middle of the blackboard
- write on the blackboard (occasionally)
- tell the kids what to do during class

7.4.9 DoAssemblyDuty

The `DoAssemblyDuty` command is used to control Mr Withit as he does assembly duty. It makes Mr Withit do nothing (i.e. stand still) unless Eric is absent from the assembly hall, in which case it hands over control to the *FetchEric* command; when that command exits, the command list is restarted. The `DoAssemblyDuty` command itself exits when assembly is over (i.e. Mr Wacker has finished speaking).

The command takes two arguments that inform Mr Withit when to start and when to stop assembly duty:

- the signal that indicates when assembly has started
- the signal that indicates when assembly has finished

These signals are raised by Mr Wacker.

7.4.10 DropStinkbomb

The `DropStinkbomb` command is used internally to make a stinkbomb-carrying character (i.e. Eric) drop a stinkbomb (Back to Skool only).

7.4.11 DumpWaterPistol

The `DumpWaterPistol` command is used internally to make Eric throw away the water pistol. The water pistol will be relocated in a random desk, and will contain water (as opposed to sherry).

7.4.12 EndGame

The `EndGame` command is used to end the game (when Eric has exceeded the lines limit, for example).

7.4.13 EvadeMouse

The `EvadeMouse` command is used internally to control a character who is scared of mice and has spotted one nearby. It makes the character either jump up and down or stand on a chair.

7.4.14 Fall

The `Fall` command is used to control the descent of a drop of water or sherry from a cup, or the descent of a conker from a tree. Until the object has been knocked out of its resting place, the command does nothing. Otherwise, it guides the object to the floor, and interacts appropriately with any character it hits. After the object has hit somebody or the floor, it is hidden from view.

7.4.15 FallToFloor

The `FallToFloor` command is used internally to control Eric's descent to the floor. It is invoked in the following situations:

- by the *RideBike* command when Eric falls off a bike that has lost momentum
- when Eric falls off a fully grown plant that has just died

7.4.16 FetchEric

The `FetchEric` command is used internally by the *ConductClassWithEric* command to make a teacher track down the truant Eric and shepherd him back to the classroom.

7.4.17 FindEric

The `FindEric` command is used to make a character look for Eric (to give him a message); it also stops the skool clock (which can be restarted later on with a *SetClock* command) to allow Eric to be found before the bell rings. When Eric has been found, he is frozen so that he has no choice but to listen to the message.

7.4.18 FindEricIfMissing

The `FindEricIfMissing` command is used by whichever teacher is on dinner duty to make him go and look for Eric if he's not in the dinner hall.

7.4.19 FindSeat

The `FindSeat` command is used to make a boy or girl find a seat in a classroom and sit down; it also makes the character find another seat if he's knocked out of one (unless the character is the swot, who must return to the same seat to avoid having to move his speech bubble during lessons).

The command takes two optional True (1) or False (0) arguments (which are both True by default). When the first argument is True, the character will seek out the back seat in the classroom first. Otherwise, when the second argument is True, the character will go to the next seat in front of him, or to the back seat if there isn't one (which is what happens when a character is pushed out of his seat). When both arguments are False, the character will sit in the seat he's standing next to (which is what happens when a character rises after being decked while seated).

7.4.20 FireCatapult

The `FireCatapult` command is used internally to make a catapult-carrying character (i.e. Eric or the tearaway) fire his catapult.

7.4.21 FireNowAndThen

The `FireNowAndThen` command is used as an argument to the *SetControllingCommand* command to make the tearaway fire his catapult occasionally. If the command decides that the time is ripe to send a projectile whizzing through the air, it passes control to the *FireCatapult* command.

7.4.22 FireWaterPistol

The `FireWaterPistol` command is used internally to make a water pistol-carrying character (i.e. Eric) fire his water pistol (Back to Skool only).

7.4.23 Flight

The `Flight` command is used internally to control Eric when he is either stepping off a fully grown plant through an open window or over the closed skool gate, or flying over the closed skool gate (after hitting it while standing on the saddle of the bike - see the *RideBike* command). The command guides Eric through his trajectory to the ground, upon which he may land with his feet, his backside, or his back.

7.4.24 Floored

The `Floored` command is used internally to control a child character who has been pushed out of his seat or knocked out cold (by Eric, the bully, or the tearaway). The command keeps the character on the floor for a brief period and then makes him stand up; after that, the character will either resume whatever he was doing before, or look for another seat (see *FindSeat*).

7.4.25 Follow

The `Follow` command is used by little boys 2-11 to sync their movements with those of the stampede leader, little boy 1. (Internally it syncs destinations, and hands over control to the *GoTo* command.) The command takes a single argument: the unique ID of the character to follow.

7.4.26 Freeze

The `Freeze` command is used internally by the *FindEric* command to freeze Eric once he has been found. It continually monitors the keyboard to check whether Eric has acknowledged delivery of a message (by pressing 'U'). It is then up to the *TellEric* or *TellEricAndWait* command to unfreeze Eric as appropriate.

7.4.27 GoTo

The `GoTo` command is arguably the most important command ever in the history of Pyskool. Without it, the characters would stay rooted to the spot, Pyskool would be boring, and you wouldn't be reading this. Sad. Anyway, `GoTo` takes a single argument, which must be one of the following:

- a skool location identifier (as found in the *[SkoolLocations]* section)
- an identifier of the form `Location:characterId`, where `characterId` is the unique ID of a character

A `Location:characterId` identifier resolves to the current location of the character with the given ID. To change the recognised prefix of such identifiers, set the `LocationMarker` configuration parameter in the *[GameConfig]* section of the ini file.

I leave it to the reader to guess what the `GoTo` command does.

7.4.28 GoToRandomLocation

The `GoToRandomLocation` command is used in many command lists to make a character go to one of his *random locations*.

7.4.29 GoTowardsXY

The `GoTowardsXY` command is used internally to make a character turn round or take one step in the direction of the destination `x` and `y` coordinates specified in the two arguments, instead of continuing all the way to the destination. The command is used primarily by the *FindEric* and *FetchEric* commands, which require tracking of a moving target (our hero) rather than a fixed destination.

7.4.30 GoToXY

The `GoToXY` command is used internally to make a character go to a location specified by an *(x, y)* pair of coordinates. In fact, the *GoTo* command resolves its location ID parameter into an *(x, y)* pair of coordinates and then hands over control to `GoToXY`. Unsurprisingly, `GoToXY` takes two arguments: *x* and *y*, as in:

```
GoToXY, 23, 17
```

Although `GoToXY` is not used explicitly in any of the stock command lists, there is nothing to stop you using it in a command list if you wish.

7.4.31 GrassAndAnswerQuestions

The `GrassAndAnswerQuestions` command is used by the swot to make him tell tales to the teacher just before class starts, and answer the teacher's questions later.

7.4.32 Grow

The `Grow` command is used to control the growth of a plant after it has been watered. If the plant is not growing, the command does nothing. Otherwise, it animates the plant growth, and lifts any characters who are standing on the plant. When the plant dies, it drops any characters who were standing on the plant, and hides it from view.

The `Grow` command takes three arguments, which specify the delay between the plant being watered and:

- appearing at half-height
- growing to full height
- dying

7.4.33 Hit

The `Hit` command is used internally to make a fist-wielding character (i.e. Eric or the bully) throw a punch.

7.4.34 HitNowAndThen

The `HitNowAndThen` command is used as an argument to the *SetControllingCommand* command to make the bully throw a punch occasionally. If the command decides that the time is ripe to send a fist whizzing through the air, it passes control to the *Hit* command.

7.4.35 Hop

The `Hop` command is used internally by the *MoveFrog* command to control the movements of a frog as it embarks on a long hop or a short hop, or turns round.

7.4.36 Jump

The `Jump` command is used internally to control Eric while he's jumping. It lifts him into the air, checks to see whether he has reached a shield, a cup, or the safe, and then lets him drop (unless there is an unconscious kid or a plant pot below).

7.4.37 JumpIfOpen

The `JumpIfOpen` command is used to jump back or forward in the command list if a door is open. The command takes two arguments: the door's unique ID (see *[Doors]*), and the number of commands to jump back or forward, as in:

```
JumpIfOpen, SkoolDoor, -5
```

7.4.38 JumpIfShut

The `JumpIfShut` command is used to jump back or forward in the command list if a door is shut. The command takes two arguments: the door's unique ID (see *[Doors]*), and the number of commands to jump back or forward, as in:

```
JumpIfShut, SkoolGate, 3
```

7.4.39 JumpOffSaddle

The `JumpOffSaddle` command is used internally to control Eric when he is jumping off the saddle of the bike (see the *RideBike* command). It lifts him into the air, checks to see whether he has reached a cup (into which a frog may be placed), and then lets him drop to the floor.

7.4.40 Kiss

The `Kiss` command is used internally to control Eric while he's kissing (or trying to kiss) Hayley. If Hayley is neither facing Eric nor in front of him at the time of the attempted kiss, Eric will take a step forward and then back again, with no kiss scored. If Hayley is in front of Eric and facing him, one of two things will happen: (a) Eric will score a kiss, or (b) Hayley will smack Eric in the face (if she feels he's tried to grab one kiss too many already). If Eric does land a kiss, his lines total will be reduced by 1000 (or to zero if he has less than 1000 lines).

7.4.41 KnockedOver

The `KnockedOver` command is used internally to control an adult character who has been downed by a catapult pellet or a stampeding boy. `KnockedOver` stuns the character, makes him reveal his safe combination letter (if appropriate), also makes him give lines to the nearest main child character (if any), and then helps him up off the floor; after that, the character will resume whatever he was doing before.

7.4.42 MonitorEric

The `MonitorEric` command is used as a subcommand (set by the *SetSubcommand* command) by Miss Take; it makes her keep an eye out for Eric in the girls' skool when it's not playtime. It takes two arguments:

- the ID of the command list to switch to in order to chase Eric
- the x-coordinate beyond which Eric must be to be regarded as worth chasing

7.4.43 MoveAboutUntil

The `MoveAboutUntil` command is used to make a character repeatedly walk a random number of paces away from a fixed point (the walkabout origin) and back again. The walkabout origin is the point the character reached before `MoveAboutUntil` was invoked.

The command takes two arguments:

- a signal to listen for; when it is raised, the character will proceed to the next command in the command list
- (optional) the minimum and maximum distances to walk away from the walkabout origin; the default is `(1, 7)`

7.4.44 MoveBike

The `MoveBike` command is used to control the bike when Eric is not sitting on the saddle. If the bike has not been unchained yet, or is resting on the ground, the command does nothing. Otherwise, it moves the bike along until it runs out of momentum, at which point it will fall over.

7.4.45 MoveDeskLid

The `MoveDeskLid` command is used to control a desk lid when it has been raised (by Eric). When the desk lid is not raised, the command does nothing. Otherwise, it transfers the contents of the desk (if any) to Eric's pocket. When the desk lid is ready to drop - after the delay specified by the command's sole argument - it is hidden from view.

7.4.46 MoveFrog

The `MoveFrog` command is used to control the movements of a frog. When a frog decides to move, it chooses from three options: turn round, short hop, and long hop. Each of these movements is controlled by the *Hop* command.

The `MoveFrog` command takes three arguments, which specify the probability that the frog will:

- keep still if Eric is not nearby
- turn round (if he decides to move at all)
- attempt a short hop (instead of a long hop) if not turning round

7.4.47 MoveMouse

The `MoveMouse` command is used to control the movements of a mouse: sprint up and down a few times, hide for a brief period, repeat.

The `MoveMouse` command takes four arguments:

- the minimum and maximum delays before the mouse comes out of hiding (e.g. `(5, 20)`)
- the minimum and maximum number of sprints the mouse will make before hiding (e.g. `(2, 5)`)
- the minimum and maximum distances of a sprint (e.g. `(2, 5)`)
- the minimum and maximum number of sprint sessions the mouse will engage in before dying (if released by Eric; e.g. `(10, 41)`)

7.4.48 MovePellet

The `MovePellet` command is used to control a catapult pellet. If the pellet has not been launched, the command does nothing. Otherwise, it moves the pellet through the air, checking whether any shields or unfortunate characters lie in its path. When a pellet has finished its flight, it is hidden from view.

7.4.49 MoveWater

The `MoveWater` command is used to control a jet of water fired from a water pistol. If the water has not been fired, the command does nothing. Otherwise, it moves the water through the air, checking whether any cups or plant pots lie in its path. When the water has finished its flight, it is hidden from view.

7.4.50 OpenDoor

The `OpenDoor` command makes a character open a door. It takes one argument: the unique ID of the door (see *[Doors]*) to open. If the door is already open, the command does nothing.

7.4.51 Pause

The `Pause` command is used internally by the *Kiss* command to occupy Hayley (i.e. prevent her from executing her current command list) while she is responding to an attempted kiss from Eric. The command exits (and Hayley will resume her current command list) after the response (a kiss or slap in the face) has been made.

7.4.52 ReleaseMice

The `ReleaseMice` command is used internally to control Eric when he's releasing mice. It makes Eric bend over, releases up to five mice (depending on how many Eric has caught) at the spot in front of Eric, and then makes Eric stand up.

7.4.53 Restart

The `Restart` command is used to return to the first command in the command list.

7.4.54 RideBike

The `RideBike` command is used internally to control Eric while he's on the bike (Back to Skool only). It may hand over control to another command depending on what happens while Eric is on the bike:

- *FallToFloor* (if the bike runs out of momentum)
- *JumpOffSaddle* (if Eric jumps off the saddle)
- *Flight* (if the bike hits the closed skool gate while Eric is standing on the saddle)

7.4.55 Say

The `Say` command is used internally to make a character say something. It takes two arguments: the thing to say, and an optional second argument specifying whether to notify listeners when done (which defaults to *False*, and is set to *True* only during lessons so that the teacher and the swot don't talk over each other). For example:

```
Say, 'Hello mum!'
```

would make a character say 'Hello mum!'.

Although `Say` is not used explicitly in any of the stock command lists, there is nothing to stop you using it in a command list if you wish.

7.4.56 SetClock

The `SetClock` command restarts the skool clock with a certain amount of time remaining until the bell rings, specified by the sole parameter. It is used (for example) to ensure that the bell rings shortly after Mr Wacker has finished delivering the detention message in assembly.

7.4.57 SetControllingCommand

The `SetControllingCommand` command is an awkwardly named command that takes another command - and that command's parameters - as its arguments, as in:

```
SetControllingCommand, OtherCommand, SomeParameter
```

What happens then is that on every pass through the main loop of the game, `OtherCommand` (the 'controlling' command) will be called for the character so controlled. The idea is that `OtherCommand` will make the character do something continuously (e.g. walk fast) or occasionally (e.g. fire a catapult or throw a punch).

The 'controlling' command remains in effect until the following command in the command list has completed.

7.4.58 SetRestartPoint

The `SetRestartPoint` command has the effect of discarding itself and all previous commands in the command list, so that any *Restart* or *StartLessonIfReady* command appearing further down the command list will bring control back up the list to the command following `SetRestartPoint` instead of the top of the list.

7.4.59 SetSubcommand

The `SetSubcommand` command places a subcommand in the character's current command list. This subcommand is then executed on each pass through the main loop, before and in addition to the current command in the character's command list. The parameters of `SetSubcommand` are the subcommand name and the subcommand's parameters, as in:

```
SetSubcommand, SomeSubcommand, SomeParameter1, SomeParameter2
```

`SetSubcommand` is used (for example) by Miss Take to place the *MonitorEric* subcommand in her command list, which makes her keep an eye out for Eric in the girls' skool when it's not playtime.

The subcommand persists for the duration of the command list (which is usually until the end of the lesson).

7.4.60 ShadowEric

The `ShadowEric` command is used by Mr Wacker when he's been alerted that Eric is trying to escape (see the *WatchForEric* command). The command makes Mr Wacker track down Eric and shadow him until the bell rings.

7.4.61 ShutDoor

The `ShutDoor` command makes a character shut a door. It takes one argument: unique ID of the door (see *[Doors]*) to shut. If the door is already shut, the command does nothing.

7.4.62 Signal

The `Signal` command is used to raise a signal. Signals are used, for example, by the *MoveAboutUntil* command to make a character pace up and down until the time is right to proceed to the next command in the command list. This scheme allows characters' movements to be coordinated.

The `Signal` command takes a single argument: the name of the signal to raise.

See also the *Unsignal* command.

7.4.63 SitForAssembly

The `SitForAssembly` command makes a character find a spot to sit down in the assembly hall until the headmaster has finished speaking. The command takes three arguments:

- the signal to wait for before standing up (see *StartAssemblyIfReady*)
- the direction to face when sitting down (`-1` for left, `1` for right)
- (optional) the minimum and maximum distances the character should walk back to find a spot to sit; the default is `(1, 4)`

7.4.64 SitStill

The `SitStill` command is always found immediately after the `FindSeat` command when it appears in a command list. It makes the character stay seated (in other words, do nothing).

7.4.65 StalkAndHit

It sounds brutal, but there really was a command list in Back to Skool that contained instructions to make the bully track down Eric's girlfriend in order to knock her about. In Pyskool, the equivalent (but more flexible) command is `StalkAndHit`, which takes a single argument: the unique ID of the character to track down.

`StalkAndHit` should be used as an argument to the *SetControllingCommand* command, as in:

```
SetControllingCommand, StalkAndHit, HEROINE
```

As a controlling command, `StalkAndHit` continually updates the character's destination to match that of the target, and makes him throw punches now and then along the way.

7.4.66 StartAssemblyIfReady

The `StartAssemblyIfReady` command makes Mr Wacker return to the start of the command list unless it's time to go down to the stage for assembly, at which point the signal named by the command's sole argument will be raised (so that the kids know when to sit down; see *SitForAssembly*).

7.4.67 StartDinnerIfReady

The `StartDinnerIfReady` command is used by teachers on dinner duty. It restarts the command list unless it's time to start looking out for Eric during dinner.

7.4.68 StartLessonIfReady

The `StartLessonIfReady` command is used by teachers to get a lesson under way (if enough time has passed since the bell rang). The command takes a single argument: the name of the signal that indicates which room the teacher will teach in (which is listened for by the kids in the classroom so that they know when to sit down). If it's not time to start the lesson yet, the command list is restarted.

7.4.69 Stink

The `Stink` command is used to control a stinkbomb after it's been dropped. If the stinkbomb has not been dropped, the command does nothing. Otherwise, it animates the stinkbomb cloud, checking whether any characters with a sensitive nose are nearby, and compelling them to open the nearest window. When the stench has dissipated - after a period specified by the command's sole argument - the cloud is hidden from view.

7.4.70 StopEric

The `StopEric` command is used internally by the *WatchForEric* command to make Albert raise his arm and alert Mr Wacker when he has spotted Eric trying to escape. The command exits when Eric leaves the 'danger zone' near Albert.

7.4.71 TellClassWhatToDo

The `TellClassWhatToDo` command is used internally by the *ConductClassWithEric* and *ConductClassWithoutEric* commands to make a teacher tell the class what to do (which usually involves writing an essay, turning to a certain page in their books, or revising for their exams).

7.4.72 TellEric

The `TellEric` command is used to make a character deliver the message specified in the command's sole argument, and then unfreeze Eric (if he was frozen, as by the *FindEric* command).

7.4.73 TellEricAndWait

The `TellEricAndWait` command is used to make a character deliver the message specified in the command's sole argument, and then unfreeze Eric (if he was frozen, as by the *FindEric* command) as soon as he has registered understanding of the message so delivered. If Eric is slow to respond, the message will be repeated periodically.

7.4.74 TellKidsToSitDown

The `TellKidsToSitDown` command is used internally by the *StartLessonIfReady* command to make a character (a teacher, normally) tell the kids to sit down when it's time to start class.

7.4.75 TripPeopleUp

The `TripPeopleUp` command is used as an argument to the *SetControllingCommand* command to make a character trip up anyone in his path as he proceeds to his destination.

7.4.76 Unsignal

The `Unsignal` command is used to lower a signal previously raised. It takes a signal name as its sole argument.

7.4.77 WaitAtDoor

The `WaitAtDoor` command is used to make Albert wait at the skool door or the skool gate until all the characters are on the correct side and it's therefore safe to shut the door or gate. The character flags *B* and *G* (see *[Characters]*) are used to determine which skool (and hence which side of the door) a character belongs to. The `WaitAtDoor` command takes a single argument: the unique ID of the door or gate (see *[Doors]*).

7.4.78 WaitUntil

The `WaitUntil` command is used to make a character do nothing (i.e. stand still) until a signal is raised. The command takes a single argument: the name of the signal to wait for.

7.4.79 WalkAround

The `WalkAround` command makes a character walk up and down about a fixed point (the walkabout origin).

The command takes two arguments:

- the number of walkarounds to do - a "walkaround" being a short trip away from the walkabout origin (wherever the character was when the `WalkAround` command was invoked) and back again
- (optional) the minimum and maximum distances to walk away from the walkabout origin; the default is `(1, 7)`

The `WalkAround` command is also used internally by the *MoveAboutUntil* command.

7.4.80 WalkFast

The `WalkFast` command is used as an argument to *SetControllingCommand* to make a character walk fast.

7.4.81 WalkUpOrDown

The `WalkUpOrDown` command is used internally by the *ConductClassWithEric* and *ConductClassWithoutEric* commands to make a teacher turn round and walk three paces. Called repeatedly, it makes the teacher walk up and down.

7.4.82 WatchForEric

The `WatchForEric` command is used as an argument to *SetControllingCommand* to make Albert keep his eyes peeled for our hero jumping out of skool windows. If Albert does spot Eric trying to escape, control is handed over to the *StopEric* command.

The `WatchForEric` command takes five arguments

- the ID of the character who will be alerted by Albert when he spots Eric trying to escape
- the ID of the command list to use for the character who will be alerted
- the alert message to be screamed by Albert
- the x-coordinate beyond which Eric should be regarded as trying to escape (96 in the stock Pyskool, which is the x-coordinate of the boys' skool door)
- the minimum and maximum distances to the left of Albert that Eric would have to be between for Albert to raise the alarm

7.4.83 WipeBoard

The `WipeBoard` command is used internally by the *ConductClassWithEric* and *ConductClassWithoutEric* commands to make a character wipe a blackboard clean.

7.4.84 Write

The `Write` command is used internally to control Eric while he's writing on a blackboard. It would be of no use in a command list.

7.4.85 WriteOnBoard

The `WriteOnBoard` command is used internally by the *ConductClassWithEric*, *ConductClassWithoutEric* and *WriteOnBoardUnless* commands to make a character write on a blackboard. The character should (ideally) be standing at the target blackboard before this command is invoked.

The command takes a single argument, namely the message to be written on the board. So if you wanted to use the command explicitly in a command list, you could put something like:

```
GoTo, ExamRoomBlackboard:Middle
WriteOnBoard, 'Pyskool rox!'
```

7.4.86 WriteOnBoardUnless

The `WriteOnBoardUnless` command is used by the tearaway to make him write on a blackboard unless the board has already been written on or the signal named in the command's sole argument has been raised.

CHAPTER 8

API documentation

- *genindex*
- *modindex*

8.1 ai

Classes that implement the commands found in command lists, such as `GoTo` and `ConductClass`.

class `ai.`**`AddLines`**(*lines*)
Command that adds lines to Eric's total. May be used by any character.

Parameters **lines** (*number*) – The lines to add to Eric's total.

`execute`()
Make a character add lines to Eric's total.

Returns *self*.

`is_interruptible`()
Return whether this command is interruptible.

Returns *False*.

class `ai.`**`Catch`**
Command that makes Eric catch a mouse or frog (if one is at hand).

`catch_animal`()
Make Eric catch a mouse or frog if one is at the location of his hand.

Returns *None*.

`get_commands`()
Return the list of steps (methods) to execute for this complex command. The steps are:

- `catch_animal()`
- `stand_up()`

`stand_up`()
Make Eric stand up straight after bending over to catch a mouse or frog.

Returns *self*.

class `ai.`**`ChaseEricOut`**(*min_x*)
Command that makes a character chase Eric up to the girls' side of the skool gate. It is used by Miss Take if she spots him in the girls' skool when it's not playtime.

Parameters **min_x** – The x-coordinate at which the character should stop chasing Eric, and stand on guard.

execute()
Make a character take the next step while chasing Eric, or remain by his side if already there.

Returns A `GoTowardsXY` command if the character hasn't caught up with Eric yet, or *None* if the character is already by Eric's side.

class ai.**CheckIfTouchingEric**(*eric_knows_signal*, *eric_has_mumps_signal*)
Command that checks whether a character is touching Eric. This command is used by Angelface when he has mumps.

Parameters

- **eric_knows_signal** – The signal used to indicate that Eric has been told to avoid the character who has mumps.
- **eric_has_mumps_signal** – The signal to use to indicate that Eric has mumps.

execute()
Make a character check whether he's touching Eric (and has therefore transmitted his disease). If Eric has been informed of the character's condition, and the character is touching Eric, the appropriate signal is raised to indicate that Eric has mumps.

Returns *self*.

class ai.**Command**
Abstract class from which all other command classes inherit. Subclasses should implement a method named *execute*.

execute()
Execute the command. The default implementation provided here does nothing; subclasses should override this method to supply the desired behaviour.

Returns *self*

finish()
Perform any required cleanup before the command is removed from the command stack. This method is called on a controlling command just before it is terminated, and on an interruptible command when it is interrupted. The default implementation provided here does nothing; subclasses override this method as necessary.

is_GoTo()
Return whether this command is one of the `GoTo` commands. This method returns *False*, but the `GoTo` commands override it and return *True*.

is_interruptible()
Return whether this command is interruptible. An interruptible command can be terminated immediately by a command list restart. This method returns *True*, but subclasses may override it and return *False* instead.

class ai.**CommandList**(*character*)
A list of commands built from a `CommandListTemplate`. Maintains a command stack from which commands are popped after they have finished executing.

Parameters **character** (`Character`) – The character to be controlled (the command list owner).

add_command(*command*)
Add a command to the stack.

Parameters **command** – The `Command` to add.

command()
Hand control of the command list owner over to the current command. The steps taken are:

1. Add the controlling command (if there is one, and the current command is interruptible) to the stack.
2. Add the subcommand (if there is one, and the current command is interruptible) to the stack.
3. If the command stack is empty, pull the next command from the command list and add it to the stack.
4. Execute the current command (the last command on the stack).
5. Act on the return value from the current command; if it is:

- *None*, then return from this method;
- *self* (the command itself), then pop the current command from the stack and go to step 3;
- another command, then add that command to the stack and go to step 4.

get_GoTo_destination()
Return the destination of the character, or *None* if he is not under the control of a `GoTo` command.

is_GoToing()
Return whether the character is under the control of one of the `GoTo` commands.

is_interruptible()
Return *False* if the command stack contains an uninterruptible command, *True* otherwise.

jump(*offset*)
Jump forwards (or backwards) in the list of commands.

Parameters **offset** – The offset by which to jump. -2 means the previous command, -1 means the current command, 0 means the next command, and 1 means the next command but one.

restart(*index=None*)
Restart this command list. When the current command finishes, the next command will be the first command in the command list, or the command denoted by *index*.

Parameters **index** – The index of the command at which to restart (defaults to 0).

set_GoTo_destination(*destination*)
Set the destination of the character if he is under the control of a `GoTo` command.

Parameters **destination** (`Location`) – The destination to set.

set_controlling_command(*command*)
Set the controlling command for this command list. The controlling command, if set, is executed before and in addition to the current command on the stack (see `command()`). `WalkFast` and `HitNowAndThen` are examples of commands that are used as controlling commands.

Parameters **command** (`Command`) – The command to set as the controlling command.

set_restart_point()
Remove the current and all previous commands from the command list. When the current command finishes, the next command will be regarded as the first for the purposes of a command list restart.

set_subcommand(*command_name*, *args*)
Set the subcommand for this command list. The subcommand, if set, is executed before and in addition to the current command on the stack (see `command()`). `MonitorEric` is an example of a command that is used as a subcommand.

Parameters

- **command_name** – The name of the command to set as the subcommand.

- **args** – The subcommand's arguments.

set_template(*template*)
Set the template for this command list. This method is used to replace a character's current command list (e.g. when the bell rings). Any interruptible commands remaining on the stack are removed (after calling their `finish()` methods); uninterruptible commands are left in place so that they have a chance to finish before the new command list kicks in.

Parameters **template** – The `CommandListTemplate` to use.

class ai.**CommandListTemplate**(*command_list_id*)
Template from which a specific command list may be created (one or more times). *command_list_id* is an identifier for the command list, used only for debugging purposes.

add_command(*command_class*, **params*)
Add a command to this template.

Parameters

- **command_class** – The class object that implements the command.
- **params** – The command's parameters.

get_commands(*start*)
Return a list of commands (initialised command class instances) constructed from this template.

Parameters **start** – The index of the first command. If *start* is 0, the entire list of commands is returned. If *start* is N>0, the first N commands are omitted.

class ai.**ComplexCommand**
Abstract class used by many commands that require a fixed sequence of steps to be executed with little or no conditional logic. Each step is implemented as a separate method. Subclasses must implement a method named *get_commands* that returns a list of the names of the methods to execute.

done()
Terminate this complex command. This method is typically used as the last step in the sequence, and is provided for convenience to subclasses.

Returns *self* (to terminate this command)

execute()
Execute the next step (method) in this command's sequence. What happens after that depends on the return value from the method called; if it is:

- *False*, then the step is taken to be still in progress, and will be executed again the next time this method is called
- *None*, then the step is taken to be finished
- *self* (this command), then the entire command is taken to be finished, and no more steps will be executed
- another command, then that command is added to the stack (so that it will be executed before proceeding to the next step)

restart()
Return to the first step in the sequence.

class ai.**ConductAssembly**
Command that makes a character conduct assembly. This involves delivering a detention message.

execute()
Make a character deliver a detention message. After the message has been delivered, a signal is raised to indicate that assembly is finished.

Returns *self* if the detention message has been delivered, or a `Say` command.

class `ai.`**`ConductClass`**(*signal=None*, *qa_group=None*)

Command that makes a character conduct a class. It determines whether the character is teaching Eric, and passes control to a `ConductClassWithEric` or `ConductClassWithoutEric` command as appropriate.

Parameters

- **signal** – The signal raised by the swot to indicate that he's ready to start the lesson.
- **qa_group** – The Q&A group from which to choose questions and answers for the teacher and the swot; if *None*, the Q&A group will be chosen at random from those available each time a question and answer is generated.

`execute`()

Make a character conduct a class.

Returns A `ConductClassWithoutEric` command if the character is not teaching Eric; *None* if the character is teaching Eric but the swot hasn't shown up yet; a `ConductClassWithEric` command otherwise.

class `ai.`**`ConductClassWithEric`**(*qa_group=None*)

Command that makes a character conduct a class with Eric. This involves waiting until the swot shows up, listening to him while he tells tales and responding appropriately, wiping the board (if there is one in the room), optionally writing on the board, and finally either telling the kids what to do and pacing up and down until the bell rings, or starting a question-and-answer session with the swot. At various points in this process, the character will also respond to the swot's tales about Eric being missing.

Parameters **qa_group** – The Q&A group from which to choose questions and answers for the teacher and the swot; if *None*, the Q&A group will be chosen at random from those available each time a question and answer is generated.

`execute`()

Make a character perform the next required action while conducting the class.

Returns *None* if the character should do nothing at the moment, or an appropriate `Command`.

class `ai.`**`ConductClassWithoutEric`**

Command that makes a character conduct a class without Eric. This involves wiping the board (if there is one in the room), optionally writing on the board, telling the kids what to do, and then pacing up and down until the bell rings.

`execute`()

Make a character perform the next step in conducting this class.

Returns A `WipeBoard` command; a `GoToXY` command to make the character return to the middle of the board after wiping it; a `WriteOnBoard` command (possibly); a `TellClassWhatToDo` command; or a `WalkUpOrDown` command.

class `ai.`**`DoAssemblyDuty`**(*assembly_started*, *assembly_finished*)

Command that makes a character perform assembly duty. This involves checking whether Eric is present in the assembly hall, and chasing him down if he's absent.

Parameters

- **assembly_started** – The signal that indicates assembly has started.
- **assembly_finished** – The signal that indicates assembly is finished.

`execute`()

Make a character perform assembly duty. If the character has just successfully herded the absent Eric to

the assembly hall, the character's command list is restarted (so that he returns to the back of the assembly hall and starts the Eric-watching process over again).

Returns *self* if the character has just herded Eric back to the assembly hall, or assembly has finished; *None* if it's not time to start keeping an eye out for Eric yet, or Eric is present; or a `FetchEric` command.

class `ai.`**DropStinkbomb**

Command that makes a character drop a stinkbomb.

drop()

Make a character drop a stinkbomb (thus creating a stinkbomb cloud).

Returns *None*.

get_commands()

Return the list of steps (methods) to execute for this complex command. The steps are:

- `raise_arm()`
- `drop()`
- `lower()`
- `done()`

is_interruptible()

Return whether this command is interruptible.

Returns *False*.

lower()

Make a character lower his arm after dropping a stinkbomb.

Returns *None*.

raise_arm()

Make a character raise his arm in preparation for dropping a stinkbomb.

Returns *None*.

class `ai.`**DumpWaterPistol**

Command that makes Eric throw away his water pistol.

dump_water_pistol()

Make Eric drop his water pistol.

Returns *None*.

get_commands()

Return the list of steps (methods) to execute for this complex command. The steps are:

- `dump_water_pistol()`
- `stand_up()`

stand_up()

Make Eric stand up straight again after dropping his water pistol.

Returns *None*.

class `ai.`**EndGame**

Command that ends the game. May be used by any character.

execute()

Make a character end the game.

Returns *None*.

class ai.**EvadeMouse**(*delay*)
Command that makes a character stand on a chair or start jumping.

Parameters **delay** – The delay before the character will get off a chair or stop jumping.

execute()
Make a character:

1. stand or remain standing on a chair, or
2. start or continue jumping, or
3. finish either one of these activities (after a certain time).

Returns *self* if the character has finished standing on a chair or jumping, *None* otherwise.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (a musophobe must not be distracted while evading a mouse).

class ai.**Fall**
Command that controls a falling object (a conker or a drop of water or sherry).

execute()
Control an object that may be falling. If the object is not falling, do nothing; otherwise make it fall a bit, taking appropriate action if it hits the floor or a person's head.

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (falling objects do not care about the bell).

class ai.**FallToFloor**
Command that controls Eric's descent to the floor (as from a bike that has run out of momentum, or a plant that has died).

execute()
Make Eric fall. If he has reached the floor, he will assume a sitting position.

Returns *self* if Eric has hit the floor, or *None* if he's still falling.

class ai.**FetchEric**
Command that makes a character find and hover around Eric until he goes to wherever he should be.

execute()
Make a character take the next step in the search for Eric.

Returns *self* if Eric is where he should be, or is due to be expelled; *None* if the character is already beside Eric; or a `GoTowardsXY` command.

class ai.**FindEric**
Command that makes a character go and find Eric.

execute()
Make a character take the next step in the search for Eric. The skool clock is stopped to ensure that the character has enough time to find him. When Eric is found, he is frozen until the character decides to unfreeze him.

Returns *self* if Eric has been found and frozen, *None* if Eric has been found but cannot be frozen at the moment, or a `GoTowardsXY` command.

class `ai.`**FindEricIfMissing**
Command that makes a character start looking for Eric if he's not where he should be.

execute()
Make a character check whether Eric is where he should be, and go looking for him if not.

Returns *self* if Eric is where he should be, or a `FetchEric` command.

class `ai.`**FindSeat**(*go_to_back=True*, *move_along=True*)
Command that makes a character find a seat and sit down.

Parameters

- **go_to_back** – *True* if the character should seek out the back seat, *False* if he should find the nearest seat.
- **move_along** – *True* if the character should move along to the next seat even if he's already standing beside one, *False* otherwise.

execute()
Make a character find a seat, turn round if he's already beside one but facing the wrong way, or sit down.

Returns *self* if the character sat down, *None* if he's beside a seat but had to turn round, or a `GoToXY` command to send him to a seat.

class `ai.`**FireCatapult**
Command that makes a character fire a catapult.

aim()
Make a character start or finish raising his catapult to eye level.

Returns *False* if the character has yet to raise the catapult to eye level, or *None* if he has raised it to eye level.

fire()
Make a character launch a catapult pellet.

Returns *None*.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- `aim()`
- `fire()`
- `lower()`
- `done()`

is_interruptible()
Return whether this command is interruptible.

Returns *False* (a catapult-firing character should finish what he's started).

lower()
Make a character start or finish lowering his catapult.

Returns *False* if the character has not finished lowering his catapult, or *None* if he has.

class `ai.`**FireNowAndThen**
Command that makes a character check whether it's a good time to launch a catapult pellet, and act accordingly.

This command is used as a controlling command (see `CommandList.set_controlling_command()`).

get_command()
Returns the command that should be used to make a character fire his catapult.

Returns A `FireCatapult` command.

ready()
Return whether this character will fire a catapult pellet. The answer will be *True* if all the conditions described in `HitOrFireNowAndThen.ready` are met, and also:

- The character's x-coordinate is divisible by 4.
- The character's catapult pellet is not currently airborne.
- The character chooses to.

class ai.**FireWaterPistol**
Command that makes a character fire a water pistol.

aim()
Make a character take out his water pistol and aim it.

Returns *None*.

fire()
Make a character pull the trigger of his water pistol (thus releasing a jet of water or sherry).

Returns *None*.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- `aim()`
- `fire()`
- `lower()`
- `done()`

is_interruptible()
Return whether this command is interruptible.

Returns *False*.

lower()
Make a character put his water pistol back in his pocket.

Returns *None*.

class ai.**Flight**(*phases*, *command_list_id=None*)
Command that controls Eric's flight through a designated sequence of locations (relative to the starting point) and animatory states.

Parameters

- **phases** – The phases of animation to proceed through.
- **command_list_id** – The ID of the command list Mr Wacker should switch to when Eric hits the ground; if not blank, Eric will be paralysed when he hits the ground.

execute()
Move Eric to the next location and phase of animation in the flight sequence.

Returns *self* if Eric has landed safely; a `Freeze` command if Eric has landed but is now immobilised (as when falling from a great height); or *None* otherwise.

class `ai.`**`Floored`**(*count*)
Command that controls a character who has been knocked to the floor.

Parameters **count** – The delay before the character should get up.

`execute`()
Make a character remain on the floor, or get up if enough time has passed.

Returns *self* if the character has got up and is ready to resume normal service, *None* otherwise.

`is_interruptible`()
Return whether this command is interruptible.

Returns *False* (the bell cannot raise characters from the floor).

class `ai.`**`Follow`**(*character_id*)
Command that makes a character follow another character.

Parameters **character_id** – The ID of the character to follow.

`execute`()
Make a character go to the same destination as another character.

Returns *self* if the destination has been reached, or a `GoTo` command.

class `ai.`**`Freeze`**
Command that controls Eric while he's frozen (as by the `FindEric` command).

`execute`()
Control Eric while he's frozen. Each time this method is called, a check is made whether Eric has acknowledged understanding of the message being delivered to him (see `TellEricAndWait`).

Returns *None*.

class `ai.`**`GoTo`**(*location_id*, *destination=None*, *go_one_step=False*)
Command that makes a character go to a location.

Parameters

- **location_id** – The ID of the location to go to (may be *None*).
- **destination** (`Location`) – The location to go to (required if *location_id* is *None*).
- **go_one_step** – *True* if this command should terminate after sending the character one step towards the destination.

`execute`()
Make a character take the next step towards his destination.

Returns *self* if the character has already reached his destination, *None* otherwise.

`is_GoTo`()
Return whether this command is one of the `GoTo` commands.

Returns *True*.

class `ai.`**`GoToRandomLocation`**
Command that makes a character go to a location chosen at random.

`execute`()
Make a character start the journey towards a randomly chosen location.

Returns A `GoTo` command.

class ai.**GoToXY**(*x*, *y*)
Command that makes a character go to a location specified by x- and y-coordinates.

Parameters

- **x** – The x-coordinate of the location to go to.
- **y** – The y-coordinate of the location to go to.

class ai.**GoTowardsXY**(*x*, *y*)
Command that makes a character take one step towards a location.

Parameters

- **x** – The x-coordinate of the location to go towards.
- **y** – The y-coordinate of the location to go towards.

class ai.**GrassAndAnswerQuestions**
Command that makes a character tell tales and answer the teacher's questions during a lesson. It is used by the swot.

execute()
Make a character tell a tale, wait for a teacher to ask a question, or answer a teacher's question.

Returns *None* if the character should do nothing at the moment, or an appropriate Command.

class ai.**Grow**(*half*, *full*, *die*)
Command that controls a plant.

Parameters

- **half** – The delay between being watered and appearing at half-height.
- **full** – The delay between being watered and growing to full height.
- **die** – The delay between being watered and dying.

execute()
Control a plant. If the plant has recently been watered, it will grow to full height and then die (disappear).

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (plants do not care about the bell).

class ai.**Hit**
Command that makes a character throw a punch.

aim()
Make a character start or finish raising his fist.

Returns *None* if the character's fist is fully raised, *False* otherwise.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- aim()
- hit()
- lower()
- done()

hit()
Make a character deck anyone unfortunate enough to come into contact with his raised fist.

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False*.

lower()
Make a character start or finish lowering his fist.

Returns *None* if the character's fist is fully lowered, *False* otherwise.

class ai.**HitNowAndThen**
Command that makes a character check whether it's a good time to throw a punch, and act accordingly. This command is used as a controlling command (see `CommandList.set_controlling_command()`).

get_command()
Returns the command that should be used to make a character throw a punch.

Returns A `Hit` command.

ready()
Return whether this character will throw a punch. The answer will be *True* if all the conditions described in `HitOrFireNowAndThen.ready()` are met, and also:

•There is someone punchable in front of the character.

•The character chooses to.

class ai.**HitOrFireNowAndThen**
Abstract command that makes a character check whether it's a good time to throw a punch or launch a catapult pellet, and act accordingly.

execute()
Make a character start throwing a punch or launching a catapult pellet if conditions are favourable.

Returns *self* if conditions are not favourable, or a `Hit` or `FireCatapult` command.

ready()
Return whether now is a good time for this character to throw a punch or fire a catapult pellet. The answer will be *True* if all the following conditions are met:

•The character is not on a staircase.

•There are no adults nearby facing the character.

•The character is standing upright.

class ai.**Hop**(*phases*)
Command that controls a frog while it's hopping.

Parameters **phases** – The phases of animation to use for the hop.

execute()
Move a frog to the next phase of animation in the hop.

Returns *self* if the hop is finished, or *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False*.

class ai.**Jump**
Command that makes a character jump.

down()
Make a character return to the floor after jumping.

Returns *None*.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- up()
- down()
- done()

up()
Make a character jump into the air.

Returns *None*.

class ai.**JumpIfOpen**(*door_id*, *offset*)
Command that jumps forwards or backwards in a character's command list if a specified door or window is open.

Parameters

- **door_id** – The ID of the door or window to check.
- **offset** (*number*) – The offset by which to jump in the command list.

execute()
Jump forwards or backwards in a character's command list if a specified door or window is open.

Returns *self*.

class ai.**JumpIfShut**(*door_id*, *offset*)
Command that jumps forwards or backwards in a character's command list if a specified door or window is shut.

Parameters

- **door_id** – The ID of the door or window to check.
- **offset** (*number*) – The offset by which to jump in the command list.

execute()
Jump forwards or backwards in a character's command list if a specified door or window is shut.

Returns *self*.

class ai.**JumpOffSaddle**
Command that controls Eric after he's jumped off the saddle of the bike.

check_cup()
Place the frog in a cup (if Eric has it and has reached a cup).

Returns *None*

fall()
Guide Eric back to the floor after jumping off the saddle of the bike.

Returns *False* if Eric has not reached the floor yet, or *None* otherwise.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- `rise()`
- `reach()`
- `check_cup()`
- `fall()`
- `done()`

reach()
Make Eric rise again and raise his arm (to reach for a cup).

Returns *None*.

rise()
Make Eric rise off the bike saddle.

Returns *None*.

class ai.**Kiss**
Command that controls Eric while he kisses (or attempts to kiss) another character.

finish_kiss()
Control Eric as he finishes a kiss or attempted kiss. If there was no one within kissing range in front of Eric, he will return from the midstride position. If there was someone kissable within kissing range, they will either finish the kiss with Eric (if they accepted it) or deck him (if they decided to slap him instead).

Returns *None*.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- `start_kiss()`
- `finish_kiss()`
- `done()`

start_kiss()
Control Eric as he begins an attempt to kiss someone. If there is no one within kissing range in front of Eric, he will move midstride. If there is someone kissable within kissing range, they will either accept the kiss or slap him.

Returns *None*.

class ai.**KnockedOver**(*delay*, *reprimand_delay*, *sleep*)
Command that controls an adult character who has been knocked over by a catapult pellet or conker.

Parameters

- **delay** – The delay before the character rises.
- **reprimand_delay** – The delay before the character gives lines to someone for knocking him over.
- **sleep** – Whether the character should remain unconscious for a while (as when Albert is struck by a conker).

execute()
Control an adult character who has been knocked over. If conditions are right and the character holds a safe combination letter, he will reveal it. If enough time has passed since being knocked over, the character will give lines if any suitable recipient is nearby.

Returns *self* if the character has got up and is ready to resume normal service, *None* otherwise.

class ai.**MonitorEric**(*command_list_id*, *chase_x*)
Command that makes a character keep an eye out for Eric, and act appropriately if he's spotted. It is used by Miss Take to make her chase Eric out of the girls' skool if she spots him there when it's not playtime. This command is used as a subcommand (see `CommandList.set_subcommand()`).

Parameters

- **command_list_id** – The ID of the command list to execute if Eric is spotted.
- **chase_x** – The x-coordinate beyond which Eric must be for the monitor to start chasing Eric.

execute()
Make a character check whether they can see Eric in the girls' skool during a non-playtime period, and switch to an appropriate command list if so.

Returns *self*.

class ai.**MoveAboutUntil**(*signal*, *walkabout_range=(1*, *7)*)
Command that makes a character walk up and down until a signal is raised.

Parameters

- **signal** – The signal to wait for.
- **walkabout_range** – The minimum and maximum distances to walk away from the walkabout origin.

execute()
Make a character walk up and down unless the signal has been raised.

Returns *self* if the signal has been raised, or a `WalkAround` command.

class ai.**MoveBike**
Command that controls a bike when Eric's not sitting on the saddle.

execute()
Control a bike when Eric's not sitting on the saddle. If the bike is resting on the floor or is not yet visible, do nothing; otherwise move the bike forwards.

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (the bike does not care about the bell).

class ai.**MoveDeskLid**(*delay*)
Command that controls a desk lid.

Parameters **delay** – The delay before an opened desk lid closes.

execute()
Control a desk lid. If the desk lid is not raised, nothing happens. If the desk lid has just been raised, the contents of the desk (if any) are delivered to the lid-raiser.

Returns *None*.

class ai.**MoveDoor**(*barrier_id*, *shut*)
Abstract command that makes a character open or close a door or window.

Parameters

- **barrier_id** – The ID of the door or window.
- **shut** – *True* if the door or window should be shut, *False* otherwise.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- raise_arm()
- move_door()
- lower_arm()
- done()

is_interruptible()
Return whether this command is interruptible.

Returns *False*.

lower_arm()
Make a character lower his arm after opening or closing a door or window.

Returns *None*.

move_door()
Open or close a door or window.

Returns *None*.

raise_arm()
Make a character raise his arm in preparation for opening or closing a door or window, or do nothing if the door/window is already in the desired state.

Returns *self* if the door/window is already in the desired state, or *None* if the character raised his arm.

class ai.**MoveFrog**(*p_hop*, *p_turn_round*, *p_short_hop*)
Command that controls a frog.

Parameters

- **p_hop** – Probability that the frog will keep still if Eric is not nearby.
- **p_turn_round** – Probability that the frog will turn round.
- **p_short_hop** – Probability that the frog will attempt a short hop (instead of a long hop) if not turning round.

execute()
Control a frog.

Returns A Hop command, or *None* if the frog decides not to move.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (frogs do not care about the bell).

class ai.**MoveMouse**(*hide_range*, *sprints*, *sprint_range*, *life_range*)
Command that controls a mouse.

Parameters

- **hide_range** (*2-tuple*) – Minimum and maximum delays before the mouse comes out of hiding.
- **sprints** (*2-tuple*) – Minimum and maximum number of sprints the mouse will make before hiding.
- **sprint_range** (*2-tuple*) – Minimum and maximum distances of a sprint.

- **life_range** (*2-tuple*) – Minimum and maximum number of sprint sessions the mouse will engage in before dying (if released by Eric).

execute()
Control a mouse. The pattern of movements of a mouse is as follows:

1. Sprint up and down a few times.
2. Hide for a bit.
3. Go to 1, or die.

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (mice do not care about the bell).

class ai.**MovePellet**
Command that controls a catapult pellet.

execute()
Move a catapult pellet if it's currently airborne, remove it from sight if it's reached the end of its flight, or else do nothing. If the pellet is airborne and hits a wall, door, window, shield, cup, conker or head, appropriate action is taken.

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (a catapult pellet should not be stopped by the bell).

class ai.**MoveWater**
Command that controls a stream of liquid (water or sherry) fired from a water pistol.

execute()
Move a stream of water or sherry one phase further in its trajectory. If the liquid hits a cup, a plant or the floor on its journey, appropriate action is taken.

Returns *None*.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (flying liquids are unstoppable).

class ai.**OpenDoor**(*barrier_id*)
Command that makes a character open a door or window.

Parameters **barrier_id** – The ID of the door or window.

class ai.**Pause**
Command used to occupy a character while they are responding to an attempted kiss from Eric.

execute()
Occupy a character while they are responding to an attempted kiss from Eric. The character's animation is controlled by the `Kiss` command while this command is in effect.

Returns *self* if the character has finished responding to the kiss, or *None* if they are still occupied.

class ai.**ReleaseMice**
Command that makes Eric release some mice.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- release_mice()
- stand_up()

release_mice()
Make Eric release some mice.

Returns *None*.

stand_up()
Make Eric stand up straight after bending over to release some mice.

Returns *self*.

class ai.**Restart**
Command that restarts a character's command list.

execute()
Restart a character's command list.

Returns *self*

class ai.**RideBike**(*bike*)
Command that controls Eric while he's on a bike.

Parameters **bike** – The bike Eric's riding.

execute()
Control Eric while he's on a bike. Keep the bike moving if Eric pedals, or carry Eric along with the bike if he's standing on the saddle.

Returns *self* if Eric has dismounted; a JumpOffSaddle command if Eric jumped while standing on the saddle; a Flight command if Eric hit the skool gate while standing on the saddle; a FallToFloor command if the bike ran out of momentum; or *None* otherwise.

class ai.**Say**(*words*, *notify=False*)
Command that makes a character say something.

Parameters

- **words** – The words to say.
- **notify** – *True* if the character who is listening should be notified (as during a question-and-answer session between a teacher and the swot), *False* otherwise.

execute()
Make a character utter the next bit of whatever he's saying. If the character has finished speaking, his speech bubble is removed, and any listeners are notified.

Returns *self* if the character has finished speaking, *None* otherwise.

finish()
Remove the character's speech bubble. This method is called if the command is removed from the command stack before exiting normally.

class ai.**SetClock**(*ticks*)
Command that makes a character set the skool clock to a specified time.

Parameters **ticks** (*number*) – The time to set the clock to (remaining ticks till the bell rings).

execute()
Make a character set the skool clock.

Returns *self*.

class ai.**SetControllingCommand**(*command_name*, **params*)
Command that sets a controlling command on a character's command list. See `CommandList.set_controlling_command()` for more details.

Parameters

- **command_name** – The name of the controlling command.
- **params** – The controlling command's initialisation parameters.

execute()
Set the controlling command on a character's command list.

Returns *self*.

class ai.**SetRestartPoint**
Command that makes the next command in a character's command list be regarded as the first (for the purposes of a restart).

execute()
Make the next command in a character's command list be regarded as the first (for the purposes of a restart).

Returns *self*.

class ai.**SetSubcommand**(*command_name*, **args*)
Command that sets a subcommand on a character's command list. See `CommandList.set_controlling_command()` for more details.

Parameters

- **command_name** – The name of the subcommand.
- **args** – The subcommand's parameters.

execute()
Set a subcommand on a character's command list.

Returns *self*.

class ai.**ShadowEric**
Command that makes a character run after Eric and then hover around him until the bell rings.

execute()
Make a character take the next step in the search for Eric, or remain by his side.

Returns A `GoTowardsXY` command if the character hasn't found Eric yet, or *None* if the character is already by Eric's side.

class ai.**ShutDoor**(*barrier_id*)
Command that makes a character close a door or window.

Parameters **barrier_id** – The ID of the door or window.

class ai.**Signal**(*signal*)
Command that makes a character raise a signal.

Parameters **signal** – The signal to raise.

execute()
Make a character raise a signal.

Returns *self*.

class ai.**SitForAssembly**(*assembly_finished*, *sit_direction*, *sit_range=(1, 4)*)

Command that makes a character find a spot to sit down during assembly, and keeps him seated until assembly has finished.

Parameters

- **assembly_finished** – The signal that indicates assembly is finished.
- **sit_direction** – The direction to face when sitting down.
- **sit_range** – The minimum and maximum distances the character should walk back to find a spot to sit.

find_spot_to_sit()

Make a character find a spot to sit during assembly. If assembly has already finished, the command is terminated.

Returns *self* if assembly has finished, or a `GoToXY` command to send the character to a place to sit down.

get_commands()

Return the list of steps (methods) to execute for this complex command. The steps are:

- `find_spot_to_sit()`
- `sit_down()`
- `get_up()`
- `done()`

get_up()

Make a character get up if assembly has finished. If assembly has not finished yet, return here next time.

Returns *False* if assembly is still in progress, or *None* if the character got up off the floor.

sit_down()

Make a character sit down for assembly, or turn round to face the right way for assembly. If assembly has already finished, the command is terminated.

Returns *self* if assembly has finished, *False* if the character was turned round, or *None* if the character sat down.

class ai.**SitStill**

Command that makes a character do nothing. It is used to make a character sit still during a lesson.

execute()

Make a character do nothing.

Returns *None*.

class ai.**StalkAndHit**(*character_id*)

Command that makes a character seek out another character while throwing occasional punches. This command is used as a controlling command (see `CommandList.set_controlling_command()`).

Parameters **character_id** – The ID of the character to stalk.

execute()

Make a character proceed one step further towards whoever he's stalking, and consider throwing a punch while he's at it.

Returns A `HitNowAndThen` command.

class ai.**StartAssemblyIfReady**(*signal*)
Command that restarts a character's command list unless it's time to start assembly.

Parameters **signal** – The signal to raise to indicate that assembly has started.

execute()
Restart a character's command list unless it's time to start assembly.

Returns *self* if it's time to start assembly, *None* otherwise.

class ai.**StartDinnerIfReady**
Command that makes a character start dinner if the time has come, or restarts the character's command list if it's still too early. It is used by teachers on dinner duty.

execute()
Make a character start dinner, or restart his command list if it's too early to start dinner.

Returns *self*

class ai.**StartLessonIfReady**(*signal*)
Command that makes a character start a lesson if the time has come, or restarts the character's command list if it's still too early.

Parameters **signal** – The signal to raise when starting the lesson.

execute()
Make a character start a lesson, or restart his command list if it's too early to start the lesson. If it's time to start the lesson, the character has a special safe secret question, and he can see the answer on a nearby blackboard, he will reveal his safe secret.

Returns A `TellKidsToSitDown` command if it's time to start the lesson, *self* otherwise.

class ai.**Stink**(*delay*)
Command that controls a stinkbomb cloud.

Parameters **delay** – The delay before the stinkbomb disappears.

execute()
Animate a stinkbomb cloud (if it's visible).

Returns *None*.

class ai.**StopEric**(*alertee_id*, *command_list_id*, *alert_message*, *escape_x*, *danger_zone*)
Command that makes a character try to stop Eric escaping from skool, and alert whoever should be alerted.

Parameters

- **alertee_id** – The ID of the character to alert.
- **command_list_id** – The ID of the command list the alerted character should use.
- **alert_message** – The alert message to scream.
- **escape_x** – The x-coordinate beyond which Eric should be regarded as trying to escape.
- **danger_zone** – The minimum and maximum distance to the left of the watcher that Eric must be for him to raise the alarm.

execute()
Make a character raise the alarm (if he hasn't already) that Eric is trying to escape, and continue trying to stop Eric if need be.

Returns *self* if the character has deemed it no longer necessary to try and stop Eric escaping, or *None* otherwise.

class ai.**TellClassWhatToDo**
Command that makes a character tell a class what to do. It is used by teachers.

execute()
Make a character tell the class what to do.

Returns A Say command, or *self* if the character has finished telling the class what to do.

class ai.**TellEric**(*message*)
Command that makes a character say something to Eric.

Parameters **message** – The message to give to Eric.

execute()
Make a character say something to Eric. When the character has finished speaking, Eric will be unfrozen.

Returns *self* if the character has finished speaking, or a Say command.

class ai.**TellEricAndWait**(*message*)
Command that makes a character say something to Eric, and wait for a response.

Parameters **message** – The message to give to Eric.

execute()
Make a character say something to Eric, and then wait for a response. When Eric has responded, he will be unfrozen. If Eric does not respond within a certain period, the character will repeat the message and wait again.

Returns *self* if Eric has responded, *None* if the character is waiting for Eric to respond, or a Say command.

class ai.**TellKidsToSitDown**
Command that makes a character tell the kids to sit down.

execute()
Make a character tell the kids to sit down.

Returns A Say command, or *self* if the character has finished telling the kids to sit down.

class ai.**TripPeopleUp**
Command that makes a character trip people up while running towards his destination. This command is used as a controlling command (see CommandList.set_controlling_command()).

execute()
Make a character start or continue running, and trip up anyone in his path.

Returns *self*.

class ai.**Unsignal**(*signal*)
Command that makes a character lower a signal.

Parameters **signal** – The signal to lower.

execute()
Make a character lower a signal.

Returns *self*.

class ai.**WaitAtDoor**(*door_id*)
Command that makes a character wait at a door until everyone is on the correct side of it (that is, boys on the boys' side, girls on the girls' side).

Parameters **door_id** – The ID of the door to wait at.

execute()
Make a character wait at a door until everyone is on the correct side of it.

Returns *self* if everyone is on the correct side of the door, *None* otherwise.

class ai.**WaitUntil**(*signal*)
Command that makes a character wait until a signal is raised.

Parameters **signal** – The signal to wait for.

execute()
Make a character wait for a signal.

Returns *self* if the signal has been raised, *None* otherwise.

class ai.**WalkAround**(*walkabouts*, *walkabout_range=(1*, *7)*)
Command that makes a character walk up and down a given number of times.

Parameters

- **walkabouts** – The number of times to walk up and down.
- **walkabout_range** – The minimum and maximum distances to walk away from the walkabout origin.

execute()
Make a character walk up (away from the walkabout origin) or down (back to the walkabout origin). If the character is on a staircase, make him finish going up or down it first.

Returns *self* if the designated number of walkabouts has been performed, *None* if the character is on a staircase, or a `GoToXY` command.

class ai.**WalkFast**
Command that makes a character walk fast (as kids do half the time, and teachers do when chasing Eric). This command is used as a controlling command (see `CommandList.set_controlling_command()`).

execute()
Ensure that a character starts or continues to walk fast.

Returns *self*.

finish()
Trigger a speed change for the character so that he no longer runs continuously.

class ai.**WalkUpOrDown**
Command that makes a character walk up (away from an origin) or down (back to the origin).

execute()
Make a character walk up or down.

Returns A `GoToXY` command, or *self* if the character is ready to turn round.

class ai.**WatchForEric**(*alertee_id*, *command_list_id*, *alert_message*, *escape_x*, *danger_zone*)
Command that makes a character keep an eye out for Eric, and alert someone if they spot him trying to escape from skool. This command is used as a controlling command (see `CommandList.set_controlling_command()`).

Parameters

- **alertee_id** – The ID of the character to alert.
- **command_list_id** – The ID of the command list the alerted character should use.
- **alert_message** – The message to scream if Eric is spotted trying to escape.
- **escape_x** – The x-coordinate beyond which Eric should be regarded as trying to escape.

- **danger_zone** – The minimum and maximum distance to the left of the watcher that Eric must be for him to raise the alarm.

execute()
Make a character check whether Eric is trying to escape from skool, and take appropriate action if he is.

Returns A `StopEric` command if Eric is spotted trying to escape, or *self* otherwise.

class ai.**WipeBoard**
Command that makes a character wipe a blackboard.

get_commands()
Return the list of steps (methods) to execute for this complex command. The steps are:

- `walk()`
- `wipe()`
- `lower_arm()`
- `done()`

is_interruptible()
Return whether this command is interruptible.

Returns *False* (blackboards should never be left partially wiped).

lower_arm()
Make a character lower his arm after wiping a bit of a blackboard. If there are any bits of the blackboard left to wipe, the command is restarted.

Returns *None*.

walk()
Make a character walk to the next part of the blackboard that needs wiping. If the character hasn't started wiping yet, he is sent to the nearest edge of the blackboard.

Returns A `GoToXY` command to send the character to the appropriate location.

wipe()
Make a character raise his arm and wipe a bit of a blackboard.

Returns *None*.

class ai.**Write**
Command that controls Eric while he's writing on a blackboard.

execute()
Control Eric while he's writing on a blackboard.

Returns *self* if Eric has finished writing on the blackboard, *None* otherwise.

class ai.**WriteOnBoard**(*message*)
Command that makes a character write on a blackboard.

Parameters **message** – The message to write on the blackboard.

execute()
Make a character write the next letter of the message on a blackboard. The character's arm will be alternately raised or lowered while writing.

Returns *self* if the character has finished writing, *None* otherwise.

is_interruptible()
Return whether this command is interruptible.

Returns *False* (a blackboard should not be left half-written on)

class ai.**WriteOnBoardUnless**(*signal*)
Command that makes a character write on a blackboard unless a specified signal has been raised.

Parameters **signal** – The signal to check before writing on the blackboard.

execute()
Make a character write on a blackboard unless a signal has been raised.

Returns *self* if the signal has been raised, or the blackboard is dirty, or the character has finished writing on the blackboard; a `WriteOnBoard` command otherwise.

ai.**get_command_class**(*command_name*)
Return the class object for a given command.

Parameters **command_name** – The name of the command.

8.2 animal

8.3 animatorystates

The animatory state IDs recognised by Pyskool. They are used in the *[SpriteGroup ...]* sections of the ini file.

animatorystates.**ARM_UP** = **'ARM_UP'**
Character with arm raised (as if writing on a blackboard or opening a door).

animatorystates.**BENDING_OVER** = **'BENDING_OVER'**
Character bending over (as if to catch a mouse).

animatorystates.**BIKE_ON_FLOOR** = **'BIKE_ON_FLOOR'**
The bike lying on the floor.

animatorystates.**BIKE_UPRIGHT** = **'BIKE_UPRIGHT'**
The bike upright.

animatorystates.**CATAPULT0** = **'CATAPULT0'**
Character firing a catapult (phase 1).

animatorystates.**CATAPULT1** = **'CATAPULT1'**
Character firing a catapult (phase 2).

animatorystates.**CONKER** = **'CONKER'**
Conker.

animatorystates.**DESK_EMPTY** = **'DESK_EMPTY'**
Desk lid and empty desk.

animatorystates.**DESK_STINKBOMBS** = **'DESK_STINKBOMBS'**
Desk lid and stinkbombs.

animatorystates.**DESK_WATER_PISTOL** = **'DESK_WATER_PISTOL'**
Desk lid and water pistol.

animatorystates.**FLY** = **'FLY'**
Catapult pellet.

animatorystates.**HITTING0** = **'HITTING0'**
Character throwing a punch (phase 1).

animatorystates.**HITTING1 = 'HITTING1'**
Character throwing a punch (phase 2).

animatorystates.**HOP1 = 'HOP1'**
Frog hopping (phase 1).

animatorystates.**HOP2 = 'HOP2'**
Frog hopping (phase 2).

animatorystates.**KISSING_ERIC = 'KISSING_ERIC'**
Character kissing Eric.

animatorystates.**KNOCKED_OUT = 'KNOCKED_OUT'**
Character lying on back (as if knocked out).

animatorystates.**KNOCKED_OVER = 'KNOCKED_OVER'**
Character sitting on the floor (as when hit by a catapult pellet).

animatorystates.**PLANT_GROWING = 'PLANT_GROWING'**
Plant half-grown.

animatorystates.**PLANT_GROWN = 'PLANT_GROWN'**
Plant fully grown.

animatorystates.**RIDING_BIKE0 = 'RIDING_BIKE0'**
Character riding a bike (phase 1).

animatorystates.**RIDING_BIKE1 = 'RIDING_BIKE1'**
Character riding a bike (phase 2).

animatorystates.**RUN = 'RUN'**
Mouse running.

animatorystates.**SHERRY_DROP = 'SHERRY_DROP'**
Drop of sherry (spilt from a cup).

animatorystates.**SIT = 'SIT'**
Frog sitting still.

animatorystates.**SITTING_ON_CHAIR = 'SITTING_ON_CHAIR'**
Character sitting on a chair.

animatorystates.**SITTING_ON_FLOOR = 'SITTING_ON_FLOOR'**
Character sitting on the floor (as for assembly).

animatorystates.**STINKBOMB = 'STINKBOMB'**
Stinkbomb cloud.

animatorystates.**WALK0 = 'WALK0'**
Character standing/walking (phase 1).

animatorystates.**WALK1 = 'WALK1'**
Character standing/walking (phase 2).

animatorystates.**WALK2 = 'WALK2'**
Character standing/walking (phase 3).

animatorystates.**WALK3 = 'WALK3'**
Character standing/walking (phase 4).

animatorystates.**WATER0 = 'WATER0'**
Water fired from a water pistol (phase 1).

animatorystates.**WATER1 = 'WATER1'**
Water fired from a water pistol (phase 2).

animatorystates.**WATER2 = 'WATER2'**
Water fired from a water pistol (phase 3).

animatorystates.**WATER3 = 'WATER3'**
Water fired from a water pistol (phase 4).

animatorystates.**WATER4 = 'WATER4'**
Water fired from a water pistol (phase 5+).

animatorystates.**WATERPISTOL = 'WATERPISTOL'**
Character firing a water pistol.

animatorystates.**WATER_DROP = 'WATER_DROP'**
Drop of water (spilt from a cup).

8.4 barrier

Classes representing the parts of the skool that cause obstructions, namely walls, windows and doors.

class barrier.**Barrier**(*barrier_id*, *x*, *bottom_y*, *top_y*, *climb_phases=()*, *fly_phases=()*)
Abstract superclass of all obstructions.

Parameters

- **barrier_id** – The ID of the barrier.
- **x** – The x-coordinate of the barrier.
- **bottom_y** – The y-coordinate of the bottom of the barrier.
- **top_y** – The y-coordinate of the top of the barrier.

impedes(*character*, *distance=0*, *force_shut=False*)
Return whether a character is impeded by this barrier.

Parameters

- **character** (Character) – The character to check.
- **distance** – The maximum distance in front of the character at which the barrier should be considered an obstruction.
- **force_shut** – If *True*, the barrier will be considered an obstruction even if it's open; otherwise it will be considered an obstruction only when closed.

is_door()
Return whether the barrier is a door (or window). Subclasses override this method as appropriate.

Returns *False*.

is_shut()
Return whether the barrier is shut. Subclasses override this method as appropriate.

Returns *True*.

class barrier.**Door**(*door_id*, *x*, *bottom_y*, *top_y*, *shut*, *auto_shut_delay*, *climb_phases*, *fly_phases=()*)
A door that may be opened and closed.

Parameters

- **door_id** – The ID of the door.

- **x** – The x-coordinate of the door.
- **bottom_y** – The y-coordinate of the bottom of the door.
- **top_y** – The y-coordinate of the top of the door.
- **shut** – Whether the door is shut at the start of the game.
- **auto_shut_delay** – The delay before the door shuts automatically; if zero or negative, the door will not shut automatically.
- **climb_phases** – The sequence of animation phases to use for Eric if he climbs over the door when it's shut.
- **fly_phases** – The sequence of animation phases to use for Eric if he he flies over the door after hitting it while standing on the saddle of the bike.

auto_shut()
Return whether this door should automatically shut now.

build_images()
Build the images for the barrier. This method is called after rescaling the screen or loading a saved game.

get_images()
Return a 2-tuple containing a list of images of the current state of the door, and the coordinates at which to draw the door.

is_door()
Return whether this is a door.

Returns *True*.

is_shut()
Return whether the door is shut.

move(*shut*)
Open or close the door.

Parameters **shut** – If *True*, close the door; otherwise open it.

Returns A 2-tuple containing a list of images of the current state of the door, and the coordinates at which to draw the door.

set_images(*open_images*, *shut_images*, *top_left*)
Define the images to use for the door when open or closed.

Parameters

- **open_images** – A list of the images to use when the door is open.
- **shut_images** – A list of the images to use when the door is closed.
- **top_left** – The coordinates at which to draw the image of the door.

class barrier.**Wall**(*barrier_id*, *x*, *bottom_y*, *top_y*, *climb_phases=()*, *fly_phases=()*)
A wall in the skool.

separates(*a*, *b*)
Return whether this wall blocks the view from one location to another.

Parameters

- **a** (Location) – The first location.
- **b** (Location) – The other location.

class barrier.**Window**(*window_id*, *x*, *bottom_y*, *top_y*, *shut*, *opener_coords*, *phases*, *not_a_bird*)
A window that may be opened and closed.

Parameters

- **window_id** – The ID of the window.
- **x** – The x-coordinate of the window.
- **bottom_y** – The y-coordinate of the bottom of the window.
- **top_y** – The y-coordinate of the top of the window.
- **shut** – Whether the window is shut at the start of the game.
- **opener_coords** – Where a character should stand to open or close the window.
- **phases** – The animation phases to use for Eric if he jumps out of the window.
- **not_a_bird** – The ID of the command list Mr Wacker should switch to when Eric hits the ground after jumping out of the window; if not blank, Eric will be paralysed when he hits the ground.

8.5 bike

8.6 cast

8.7 character

8.8 debug

Used for debugging Pyskool.

debug.**error**(*text*)
Print *text* (an error message) on stderr.

debug.**log**(*text*)
Print *text* (a debug message) on stderr.

8.9 desklid

8.10 droppable

8.11 eric

8.12 floor

Defines the Floor class.

class floor.**Floor**(*floor_id*, *left_x*, *right_x*, *y*)
A region of the skool regarded as a starting point from which to reach a destination (which will be on the same or another floor).

Parameters

- **floor_id** – The floor's unique ID.
- **left_x** – The x-coordinate of the left edge of the floor.
- **right_x** – The x-coordinate of the right edge of the floor.
- **y** – The y-coordinate of the floor.

below(*character*)
Return whether this floor is below a character's current location.

Parameters **character** (`Character`) – The character to check.

contains_location(*x*, *y*)
Return whether the location *(x, y)* is on this floor.

supports(*character*)
Return whether a character is on this floor.

Parameters **character** (`Character`) – The character to check.

8.13 game

8.14 graphics

8.15 iniparser

Parse a single ini file, or a directory of ini files.

class iniparser.**IniParser**(*path*)
Parses one or more ini files.

Parameters **path** – An ini file, or a directory to scan for ini files.

get_config(*pattern*)
Return a dictionary of keys and values from every section whose name matches *pattern*.

parse_section(*name*, *parse_numbers=True*, *num_elements=None*, *split=True*, *separator=', '*)
Extract the elements from every line in a section. The return value is a list of lists of elements from each line (if *split* is *True*), or a list of the lines. If the named section does not exist, an empty list is returned.

Parameters

- **name** – The name of the section.
- **parse_numbers** – If *True*, any numeric element is converted to a number; otherwise it is left as a string.
- **num_elements** – The minimum number of elements to return for each line in the section. If any line has fewer elements than this, the list of elements is padded out with *None* instances.
- **split** – If *True*, each line is split on commas; otherwise, the line is returned whole.
- **separator** – The character sequence that separates the elements in a line.

8.16 input

8.17 items

IDs of the items that can be displayed in Eric's inventory. The image specifications for each item are defined in the *[Inventory]* section of the ini file.

items.**FROG** = **'FROG'**
Captured frog.

items.**MOUSE** = **'MOUSE'**
Captured mouse.

items.**SAFE_KEY** = **'SAFE_KEY'**
Safe key.

items.**SHERRY_PISTOL** = **'SHERRY_PISTOL'**
Water pistol containing sherry.

items.**STINKBOMBS1** = **'STINKBOMBS1'**
One stinkbomb.

items.**STINKBOMBS2** = **'STINKBOMBS2'**
Two stinkbombs.

items.**STINKBOMBS3** = **'STINKBOMBS3'**
Three stinkbombs.

items.**STOREROOM_KEY** = **'STOREROOM_KEY'**
Science Lab storeroom key.

items.**WATER_PISTOL** = **'WATER_PISTOL'**
Water pistol containing water.

8.18 keys

8.19 lesson

Classes concerned with controlling what goes on during a lesson.

class lesson.**AssemblyMessageGenerator**
Generates messages to be delivered by whoever is conducting assembly. There is only one assembly message generator, shared by the whole skool.

add_message_template(*template*)
Add *template* to the generator's collection of message templates.

add_word(*group_id*, *word*)
Add a word to the generator's collection.

Parameters

- **group_id** – The name of the group to add the word to.
- **word** – The word.

generate_message()
Return a message based on a randomly chosen template and containing randomly chosen phrases.

class lesson.**Lesson**(*cast*, *swot*, *room*, *config*)

Controls the interaction between the teacher, the swot and Eric during a lesson. The various actions required by the teacher and the swot during a lesson - such as grassing on Eric for being absent, writing on the board, and asking and answering questions - are defined by individual methods on this class.

A new lesson is created by the swot when he sits down after being told to by the teacher at the classroom doorway.

Parameters

- **cast** (Cast) – The cast.
- **swot** (Character) – The swot.
- **room** (Room) – The classroom in which the lesson is taking place.
- **config** (*dict*) – Configuration parameters from the ini file.

answer_question()

Make the swot answer the teacher's question. The swot's next action will be check_eric().

ask_question()

Make the teacher ask a question. The swot's next action is set to answer_question().

Returns A Say command.

check_eric()

Make the swot tell the teacher that Eric is absent (if he is). If Eric is absent, the teacher's next action will be fetch_eric().

Returns A Say command if Eric is absent, otherwise *None*.

check_eric_initial()

Make the swot tell the teacher that Eric is absent (if he is). This method defines the swot's first action during a lesson. If Eric is absent, the teacher's next action will be fetch_eric(). The swot's next action is set to grass_for_hitting().

Returns A Say command if Eric is absent, otherwise *None*.

fetch_eric()

Make the teacher track down Eric if he is absent. The teacher may first give lines to the swot for telling tales. If Eric is present by the time this method is called (after the swot has finished telling the teacher that Eric is not in class), the teacher will give lines to Eric for being late (or for leaving early).

Returns A FetchEric command if Eric is still absent after the swot has finished speaking, otherwise *None*.

finished_speaking()

Indicate that the current actor (teacher or swot) has finished speaking.

get_question()

Return the next question for the teacher to ask in a question-and-answer session with the swot.

give_lines(*victim_id*, *message_id*)

Make the teacher give lines to the swot for telling a tale, or give lines to the subject of the swot's tale.

Parameters

- **victim_id** – The ID of the subject (may be *None*, in which case no lines will be given).
- **message_id** – The ID of the lines message.

give_lines_for_hitting()

Make the teacher give lines to the swot for telling a tale about being hit, or give lines to the subject of the tale. If the swot has not told such a tale, nothing happens.

give_lines_for_writing()
Make the teacher give lines to the swot for telling a tale about the blackboard being written on, or give lines to the subject of the tale. If the swot has not told such a tale, nothing happens. The teacher's next action is set to `wipe_board()`.

grass_for_hitting()
Make the swot tell a tale about someone hitting him (possibly). This method defines the swot's second action during a lesson. The teacher's next action is set to `give_lines_for_hitting()`. The swot's next action is set to `grass_for_writing()`.

Returns A `Say` command, or *None* if the swot decides not to tell a tale.

grass_for_writing()
Make the swot tell a tale about someone writing on the blackboard (if it was written on by Eric or the tearaway). This method defines the swot's third action during a lesson. The teacher's next action is set to `give_lines_for_writing()`.

Returns A `Say` command, or *None* if the swot decides not to tell a tale.

is_eric_absent()
Return whether Eric is absent from the classroom in which this lesson is taking place.

join(*teacher*, *qa_generator*, *qa_group*)
Make a teacher join the lesson. This method is called by the teacher when he notices that the swot has sat down.

Parameters

- **teacher** (`Character`) – The teacher.
- **qa_generator** (`QAGenerator`) – The question-and-answer generator to use.
- **qa_group** – The Q&A group from which to choose questions and answers for the teacher and the swot; if *None*, the Q&A group will be chosen at random from those available each time a question and answer is generated.

next_swot_action()
Complete any actions required of the swot, and return the next command to be executed by him, or *None* if it's not his turn to act.

next_teacher_action()
Complete any actions required of the teacher, and return the next command to be executed by him, or *None* if it's not his turn to act.

return_to_base()
Make the teacher return to the classroom after fetching Eric. The teacher's next action will be `ask_question()` (if a question-and-answer session was interrupted) or `walk_up_or_down()`.

Returns A `GoToXY` command.

switch(*action=None*)
Switch turns between the actors in this lesson (the teacher and the swot).

Parameters **action** – The next action (method to execute) for the next actor; if *None*, the next action (which may have already been set) is unchanged.

tell_class_what_to_do()
Make the teacher tell the class what to do (as opposed to starting a question-and-answer session with the swot). The teacher's next action (and base action for the remainder of the lesson) will be `walk_up_or_down()`.

Returns A `TellClassWhatToDo` command.

walk_to_board()
Make the teacher walk to the middle of the blackboard (after having wiped it). The teacher's next action will be `write_on_board()`.

Returns A `GoToXY` command.

walk_up_or_down()
Make the teacher walk up or down in front of the blackboard. This action is used during a lesson with no question-and-answer session. The swot's next action is set to `check_eric()`.

Returns A `WalkUpOrDown` command.

wipe_board()
Make the teacher wipe the board (if there is one). The teacher's next action will be `walk_to_board()`.

Returns A `WipeBoard` command if there is a blackboard, *None* otherwise.

write_on_board()
Make the teacher write on the blackboard (possibly). The teacher's next action will be the base action for this lesson (either `tell_class_what_to_do()` or `ask_question()`).

Returns A `WriteOnBoard` command if the teacher chooses to write, otherwise *None*.

class `lesson.`**QAGenerator**
Generates questions and answers for the teacher and swot to use during a lesson. Every teacher gets his own generator to keep; it is built before the game starts.

add_answer(*question_id*, *text*)
Add an answer to a question.

Parameters

- **question_id** – The ID of the question.
- **text** – The text of the answer.

add_qa_pair(*qa_group*, *word1*, *word2*)
Add a Q&A pair to a Q&A group.

Parameters

- **qa_group** – The name of the Q&A group.
- **word1** – The first word of the pair.
- **word2** – The second word of the pair.

add_question(*question_id*, *qa_group*, *text*)
Add a question to a Q&A group.

Parameters

- **question_id** – The ID of the question.
- **qa_group** – The name of the Q&A group to add the question to.
- **text** – The text of the question.

has_special_question()
Return whether the teacher has a special question. A special question is one to which the answer must be seen written on a blackboard by the teacher to make him reveal his safe combination letter.

initialise_special_answer()
Initialise the answer to the teacher's special question (if there is one). The special answer is chosen at random from the Q&A pairs in the Q&A group of the special question.

prepare_qa(*qa_group=None*)
Prepare a randomly chosen question and answer.

Parameters **qa_group** – The Q&A group from which to choose the question and answer; if *None*, the Q&A group will be chosen at random from those available.

Returns A 2-tuple containing the question and the answer.

prepare_special_qa()
Prepare the teacher's special question and answer (if any).

Returns A 2-tuple containing the question and the answer.

set_special_group(*qa_group*, *index*)
Set the Q&A group to use for the teacher's special question (if there is one).

Parameters

- **qa_group** – The name of the Q&A group.
- **index** – The index (0 or 1) of the special answer in the Q&A pair.

8.20 lines

Lines message IDs recognised by Pyskool. The text of the lines messages themselves are defined in the *[LinesMessages]* section of the ini file.

lines.**BACK_TO_SKOOL = 'BACK_TO_SKOOL'**
Eric should be back in the boys' skool by now.

lines.**BE_PUNCTUAL = 'BE_PUNCTUAL'**
Eric showed up for class while the swot was grassing on him for being absent.

lines.**COME_ALONG_PREFIX = 'COME_ALONG'**
Prefix for the lines message IDs used when Eric's teacher is fetching him.

lines.**GET_ALONG = 'GET_ALONG'**
Eric should be in the dinner hall, assembly hall, or a particular classroom by now.

lines.**GET_OFF_PLANT = 'GET_OFF_PLANT'**
Eric is standing on a plant.

lines.**GET_OUT = 'GET_OUT'**
Eric is somewhere he should never be (such as the head's study).

lines.**GET_UP = 'GET_UP'**
Eric is sitting or lying on the floor.

lines.**NEVER_AGAIN = 'NEVER_AGAIN'**
Somebody knocked a teacher over.

lines.**NO_BIKES = 'NO_BIKES'**
Eric is riding the bike inside the boys' skool.

lines.**NO_CATAPULTS = 'NO_CATAPULTS'**
Eric is firing his catapult.

lines.**NO_HITTING = 'NO_HITTING'**
Eric is throwing a punch.

lines.**NO_JUMPING = 'NO_JUMPING'**
Eric is jumping.

lines.**NO_SITTING_ON_STAIRS** = 'NO_SITTING_ON_STAIRS'
Eric is sitting on the stairs.

lines.**NO_STINKBOMBS** = 'NO_STINKBOMBS'
Eric dropped a stinkbomb.

lines.**NO_TALES** = 'NO_TALES'
The swot told a tale.

lines.**NO_WATERPISTOLS** = 'NO_WATERPISTOLS'
Eric is firing a water pistol.

lines.**NO_WRITING** = 'NO_WRITING'
Eric is writing on a blackboard.

lines.**SIT_DOWN** = 'SIT_DOWN'
Eric is standing up in class when he should be sitting down.

lines.**SIT_FACING_STAGE** = 'SIT_FACING_STAGE'
Eric is sitting down facing the wrong way in assembly.

lines.**STAY_IN_CLASS** = 'STAY_IN_CLASS'
Eric left the classroom and then returned while the swot was grassing on him for being absent.

8.21 location

Defines the Location class.

class location.**Location**(*coords*)
A location in the skool specified by a pair of coordinates.

Parameters **coords** – The coordinates of this location.

coords()
Return the coordinates of this location as a 2-tuple.

8.22 mutable

8.23 pellet

8.24 plant

8.25 room

Classes that represent the rooms in the skool and their furniture.

class room.**Blackboard**(*screen*, *top_left*, *size*, *chalk*, *skool_image*)
A blackboard in a classroom.

Parameters

- **screen** (Screen) – The screen.
- **top_left** – The coordinates of the top-left of the blackboard.
- **size** – The size (width, height) of the blackboard.

- **chalk** – The chalk colour to use when writing on the blackboard.
- **skool_image** (`Image`) – An image of the skool.

beside(*character*)
Return whether a character is standing beside the blackboard.

Parameters **character** (`Character`) – The character to check.

build_images()
Build the images for the blackboard. This method is called after rescaling the screen or loading a saved game.

clear(*blit=False*)
Mark this blackboard as clean.

Parameters **blit** – If *True*, the blackboard surface will be blitted clean too; otherwise it will be left alone.

is_dirty()
Return whether anything is written on the blackboard.

newline()
Start a new line on the blackboard.

restore()
Restore the image of this blackboard. This method is used after restoring a saved game.

shows(*text*, *in_order=True*)
Return whether the blackboard displays all the characters in a given piece of text.

Parameters

- **text** – The text to look for.
- **in_order** – If *True*, return *True* only if the order of the characters written on the board matches too.

wipe(*column*)
Wipe a column of the blackboard clean.

Parameters **column** – The column to wipe clean.

write(*char*)
Write a character on the blackboard.

Parameters **char** – The character to write.

class `room.`**Chair**(*room*, *x*)
A seat in a classroom.

Parameters

- **room** (`Room`) – The room the chair is in.
- **x** – The x-coordinate of the chair.

seat(*character*)
Mark the chair as being occupied by a character.

Parameters **character** (`Character`) – The character sitting in the chair.

vacate()
Mark this chair as vacant.

class room.**Desk**(*room*, *x*)
A desk (that can be opened) in a classroom.

Parameters

- **room** (Room) – The room the desk is in.
- **x** – The x-coordinate of the desk.

empty()
Mark the desk as empty.

insert(*item*)
Insert an inventory item (water pistol or stinkbombs) into the desk.

Parameters **item** – The ID of the item to insert.

class room.**NoGoZone**(*zone_id*, *min_x*, *max_x*, *bottom_y*, *top_y*)
A region of the skool in which Eric is never allowed to be.

Parameters

- **zone_id** – The zone's unique ID.
- **min_x** – The x-coordinate of the left edge of the zone.
- **max_x** – The x-coordinate of the right edge of the zone.
- **bottom_y** – The y-coordinate of the bottom edge of the zone.
- **top_y** – The y-coordinate of the top edge of the zone.

contains(*x*, *y*)
Return whether a given location is inside the zone.

Parameters

- **x** – The x-coordinate of the location.
- **y** – The y-coordinate of the location.

class room.**Room**(*room_id*, *name*, *top_left*, *bottom_right*, *get_along*)
A classroom or some other region of the skool that Eric is expected to show up in when the timetable demands it.

Parameters

- **room_id** – The unique ID of the room.
- **name** – The name of the room (as it should appear in the lesson box).
- **top_left** – The coordinates of the top-left corner of the room.
- **bottom_right** – The coordinates of the bottom-right corner of the room.
- **get_along** – If *True*, Eric will be told to get along if he's found in this room when the timetable doesn't say he should be in it.

add_blackboard(*screen*, *top_left*, *size*, *chalk*, *skool_image*)
Add a blackboard to the room.

Parameters

- **screen** (Screen) – The screen.
- **top_left** – The coordinates of the top-left of the blackboard.
- **size** – The size (width, height) of the blackboard.

- **chalk** – The chalk colour to use when writing on the blackboard.
- **skool_image** (`Image`) – An image of the skool.

add_chair(*x*)
Add a chair to the room.

Parameters **x** – The x-coordinate of the chair.

add_desk(*x*)
Add a desk (that can be opened) to the room.

Parameters **x** – The x-coordinate of the desk.

beside_blackboard(*character*)
Return whether a character is standing beside the blackboard in this room.

Parameters **character** (`Character`) – The character to check.

blackboard_dirty()
Return *True* if the room has a blackboard and it is dirty, *False* otherwise.

build_blackboard_images()
Build the images for the blackboard in this room (if any). This method is called after rescaling the screen or loading a saved game.

chair(*character*, *check_dir=True*)
Return the chair in this room that a character is next to, or *None* if the character is not next to a chair.

Parameters

- **character** (`Character`) – The character to check.
- **check_dir** – If *True*, return a chair only if the character is beside one and facing the right way to sit in it; otherwise disregard the direction in which the character is facing.

contains(*character*)
Return whether a character is in this room.

Parameters **character** (`Character`) – The character to check.

desk(*character*)
Return the desk in this room that a character is sitting at, or *None* if the character is not sitting at a desk.

Parameters **character** (`Character`) – The character to check.

get_blackboard_writer()
Return the character who wrote on the blackboard in this room, or *None* if either the room has no blackboard, or the blackboard is clean.

get_chair_direction()
Return the direction in which the chairs in this room are facing.

Returns -1 if the chairs are facing left, 1 if they are facing right, or *None* if there are no chairs in the room.

get_next_chair(*character*, *move_along*, *go_to_back*)
Return the chair that a character should find and sit in.

Parameters

- **character** (`Character`) – The character looking for a chair.
- **move_along** – If *True* (and *go_to_back* is False), return the next seat along if the character is currently beside one; otherwise return the seat closest to the character.

- **go_to_back** – If *True*, return the back seat in the room.

Returns A 2-tuple containing the target chair and the direction it faces (-1 or 1).

has_blackboard()
Return whether the room has a blackboard.

restore_blackboard()
Restore the image of the blackboard in this room. This method is used to draw the contents of a blackboard afresh after restoring a saved game.

wipe_blackboard()
Wipe the blackboard in the room (if any) and mark it as clean.

8.26 scoreboard

Keep track of the score, lines total and high score.

class scoreboard.**Scoreboard**(*screen*)
The scoreboard.

Parameters **screen** (Screen) – The screen on which the scoreboard is displayed.

add_lines(*addend*)
Add lines to the lines total and print it.

Parameters **addend** – The number of lines to add.

add_to_score(*addend*)
Add points to the score and print it.

Parameters **addend** – The number of points to add.

print_score_box()
Print the score, lines total and hi-score.

reinitialise()
Reinitialise the scoreboard after a game has ended. The current score becomes the new high score if necessary, the score and lines total are reset to zero, and all three numbers are printed.

8.27 skoolbuilder

Build the skool and its cast of characters.

class skoolbuilder.**SkoolBuilder**(*path*)
Builds a skool and its cast from the contents of ini files.

build_skool(*skool*)
Build a skool from the contents of the ini files.

Parameters **skool** (Skool) – The skool to build.

8.28 skool

8.29 sound

8.30 staircase

Defines the `Staircase` class.

class `staircase.`**`Staircase`**(*bottom*, *top*, *force=False*)
A staircase.

Parameters

- **bottom** – The coordinates of the bottom of the staircase.
- **top** – The coordinates of the top of the staircase.
- **force** – If *True*, the staircase must be ascended or descended when approached (like the staircase in Back to Skool that leads up from or down to the stage).

`contains`(*character*, *distance=0*)
Return whether a character is (a) on a step of this staircase, or (b) at the bottom of this staircase facing the top, or (c) at the top of this staircase facing the bottom.

Parameters

- **character** (`Character`) – The character to check.
- **distance** – The maximum distance to check in front of the character.

`contains_location`(*x*, *y*)
Return whether the location *(x, y)* is at the bottom, or at the top, or on a step of this staircase.

`supports`(*character*)
Return whether a character is on a step of this staircase.

Parameters **character** (`Character`) – The character to check.

8.31 stinkbomb

8.32 timetable

The main timetable and the skool clock.

class `timetable.`**`Timetable`**(*config*)
Represents the timetable of lessons, and the skool clock that ticks down until the bell rings.

Parameters **config** (*dict*) – Configuration parameters from the ini file.

`add_lesson`(*lesson_id*)
Add a lesson to the timetable.

Parameters **lesson_id** – The ID of the lesson.

`add_lesson_details`(*lesson_id*, *hide_teacher*, *teacher_id*, *room_id*)
Add the details of a lesson to the timetable.

Parameters

- **lesson_id** – The ID of the lesson.
- **hide_teacher** – If *True*, the teacher's name (if any) will not be printed in the lesson box for this period.
- **teacher_id** – The ID of the teacher supervising Eric for this period, or an empty string if it is unsupervised.
- **room_id** – The ID of the room in which the lesson takes place, or the name of the period (such as PLAYTIME) if it is unsupervised.

add_special_playtime(*lesson_id*)
Add a special playtime. Special playtimes do not appear in the main timetable (though they could be inserted); occasionally a normal playtime in the main timetable will be replaced by a special playtime.

Parameters **lesson_id** – The ID of the special playtime.

get_room_id()
Return the ID of the room Eric's will be expected to show up in at some point during the current period, or else the name of the period (such as PLAYTIME).

get_teacher_id()
Return the ID of the teacher supervising Eric for the current period, or an empty string if it is an unsupervised period.

hide_teacher()
Return whether the teacher's name (if any) should be displayed in the lesson box for the current period. This is generally *True* for classroom periods, and *False* for any other period (supervised or otherwise).

is_assembly()
Return whether it's Assembly.

is_playtime()
Return whether it's Playtime.

is_teaching_eric(*character*)
Return whether a character is supervising Eric during the current period.

Parameters **character** (`Character`) – The character to check.

is_time_remaining(*ticks*)
Return whether there is no more than a certain number of skool clock ticks remaining before the bell rings.

Parameters **ticks** – The number of ticks.

is_time_to_get_along()
Return whether Eric should have left the classroom he was in last period by now.

is_time_to_start_lesson()
Return whether it's time to start a lesson. When the answer is *True*, teachers will stop pacing up and down outside classroom doorways.

next_lesson()
Proceed to the next lesson in the timetable. If the next lesson is playtime and there are any special playtimes defined, one of those may be chosen as the next lesson.

reinitialise()
Reinitialise the timetable after a game has ended.

resume(*ticks*)
Start the skool clock with a certain number of ticks remaining till the bell rings. If the clock was previously stopped (see `stop()`), it will start ticking again.

Parameters **ticks** – The number of ticks.

rewind(*ticks*)
Rewind the skool clock by a number of ticks.

Parameters **ticks** – The number of ticks.

stop()
Stop the skool clock. Any further attempts to make it tick will be futile until `resume()` is called.

tick()
Advance the skool clock by one tick (unless the clock has been stopped).

Returns *True* if it's time for the bell to ring, *False* otherwise.

up_a_year()
Take appropriate action when Eric has gone up a year. This entails setting the skool clock so that half a normal lesson length remains before the bell will ring.

8.33 water

a

b

d

f

i

l

r

s

t