simfile

Release 2.1.1

Ash Garcia

Jul 31, 2023

CONTENTS

A modern simfile parsing & editing library for Python 3.

CHAPTER

ONE

INSTALLATION

simfile is available on PyPI:

```
pip3 install simfile
```

CHAPTER

TWO

QUICKSTART

Load simfiles from disk using *simfile.open()* or *simfile.load()*:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> springtime
<SSCSimfile: Springtime>
>>> with open('testdata/nekonabe/nekonabe.sm', 'r') as infile:
...     nekonabe = simfile.load(infile)
...
>>> nekonabe
<SMSimfile: >
```

Use lowercase attributes to access most common properties:

```
>>> springtime.artist
'Kommisar'
>>> springtime.banner
'springbn.png'
>>> springtime.subtitle = '(edited)'
>>> springtime
<SSCSimfile: Springtime (edited)>
```

Alternatively, use uppercase strings to access the underlying dictionary:

```
>>> springtime['ARTIST']
'Kommisar'
>>> springtime['ARTIST'] is springtime.artist
True
>>> list(springtime.keys())[:7]
['VERSION', 'TITLE', 'SUBTITLE', 'ARTIST', 'TITLETRANSLIT', 'SUBTITLETRANSLIT',
↪'ARTISTTRANSLIT']
```

Charts are stored in a list under the `.charts` attribute and function similarly to simfile objects:

```
>>> len(springtime.charts)
9
>>> chart = springtime.charts[0]
>>> chart
<SSCChart: dance-single Challenge 12>
>>> chart.stepstype
'dance-single'
>>> list(chart.keys())[:7]
['CHARTNAME', 'STEPSTYPE', 'DESCRIPTION', 'CHARTSTYLE', 'DIFFICULTY', 'METER',
↪'RADARVALUES']
```

CHAPTER

THREE

FURTHER READING

3.1 What are simfiles?

Simfiles are the primary unit of game content for music game simulators like StepMania. These files contain song metadata and some number of charts (also known as "stepcharts" or "maps") that dictate the gameplay sequence. They are accompanied by a music file and often some graphic files like banners and backgrounds.

StepMania primarily uses two simfile formats, SM and SSC. These are the two simfile formats supported by the **simfile** library.

3.1.1 What's in a simfile?

If you open a simfile in a text editor, you'll typically be greeted with something like this:

```
#VERSION:0.83;
#TITLE:Springtime;
#SUBTITLE:;
#ARTIST:Kommisar;
#TITLETRANSLIT:;
#SUBTITLETRANSLIT:;
#ARTISTTRANSLIT:;
#GENRE:;
#ORIGIN:;
#CREDIT:;
#BANNER:springbn.png;
#BACKGROUND:spring.png;
[...]
```

StepMania uses this information to determine how to display the simfile in-game. Later in the file you'll see *timing data*, which determines how to keep simfile and audio synchronized, followed by charts containing *note data*, which contains the input sequence for the player to perform. Charts are typically written by humans to follow the rhythm of the song.

3.1.2 Why do I need a library for this?

While the file format shown above seems simple, simfiles on the Internet vary greatly in formatting and data quality, and StepMania tries its hardest to support all of these files. As a result, there are numerous edge cases and undocumented features that complicate parsing of arbitrary simfiles. Here are some examples:

- Not all simfiles are encoded in UTF-8; many older files from around the world use Windows code pages instead. StepMania tries four encodings (UTF-8 followed by three code pages) in order until one succeeds.
  - *simfile.open()* and related functions do the same.
- Many simfiles, even modern ones, contain formatting errors such as malformed comments and missing semicolons. StepMania handles missing semicolons at the protocol level and emits a warning for other formatting errors.
  - *simfile.open()* and related functions offer a *strict* parameter that can be set to `False` to ignore formatting errors.
- Holds and rolls are expected to have corresponding tail notes; a head note without a tail note (or vice-versa) is an error. StepMania emits a warning and treats disconnected head notes as tap notes (and discards orphaned tail notes).
  - *group_notes()* can do the same on an opt-in basis.
- Some properties have legacy aliases, like *FREEZES* in place of *STOPS*. Additionally, keysounded SSC charts use a *NOTES2* property for note data instead of the usual *NOTES*. StepMania looks for these aliases in the absence of the regular property name.
  - Known properties on the *Simfile* and *Chart* classes do the same.
- During development of the SSC format, timing data on charts ("split timing") was an unstable experimental feature. Modern versions of StepMania ignore timing data from these unstable versions (prior to version 0.70).
  - *TimingData* ignores SSC chart timing data from these older versions too.

Even if you don't need the rich functionality of the supplementary modules and packages, the top-level *simfile* module and the *SMSimfile* and *SSCSimfile* classes its functions return are thoroughly tested and offer simple, format-agnostic APIs. While a bespoke regular expression might be sufficient to parse the majority of simfiles, the **simfile** library is designed to handle any simfile that StepMania accepts, with escape hatches for error conditions nearly everywhere an exception can be thrown.

3.2 Reading & writing simfiles

3.2.1 Opening simfiles

The top-level *simfile* module offers 3 convenience functions for loading simfiles from the filesystem, depending on what kind of filename you have:

- *simfile.open()* takes a path to an SM or SSC file.
- *simfile.opendir()* takes a path to a simfile directory. *(new in 2.1)*
- *simfile.openpack()* takes a path to a simfile pack. *(new in 2.1)*

```
>>> import simfile
>>> springtime1 = simfile.open('testdata/Springtime/Springtime.ssc')
>>> springtime2, filename = simfile.opendir('testdata/Springtime')
>>> for sim, filename in simfile.openpack('testdata'):
...     if sim.title == 'Springtime':
```

(continues on next page)

(continued from previous page)

```
...         springtime3 = sim
...
>>> print springtime1 == springtime2 and springtime2 == springtime3
True
```

Plus two more that don't take filenames:

- *simfile.load()* takes a file object.
- *simfile.loads()* takes a string of simfile data.

Note: If you're about to write this:

```
with open('path/to/simfile.sm', 'r') as infile:
    sim = simfile.load(infile)
```

Consider writing `sim = simfile.open('path/to/simfile.sm')` instead. This lets the library determine the correct encoding, rather than defaulting to your system's preferred encoding. It's also shorter and easier to remember.

The type returned by functions like *open()* and *load()* is declared as *Simfile*. This is a union of the two concrete simfile types, *SMSimfile* and *SSCSimfile*:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> type(springtime)
<class 'simfile.ssc.SSCSimfile'>
>>> nekonabe = simfile.open('testdata/nekonabe/nekonabe.sm')
>>> type(nekonabe)
<class 'simfile.sm.SMSimfile'>
```

The "magic" that determines which type to use is documented under *simfile.load()*. If you'd rather use the underlying types directly, instantiate them with either a *file* or *string* argument:

```
>>> from simfile.ssc import SSCSimfile
>>> with open('testdata/Springtime/Springtime.ssc', 'r') as infile:
...     springtime = SSCSimfile(file=infile)
```

Note: These *Simfile* types don't know about the filesystem; you can't pass them a filename directly, nor do they offer a `.save()` method (see *Writing simfiles to disk* for alternatives). Decoupling this knowledge from the simfile itself enables them to live in-memory, without a corresponding file and without introducing state-specific functionality to the core simfile classes.

3.2.2 Accessing simfile properties

Earlier we used the *title* attribute to get a simfile's title. Many other properties are exposed as attributes as well:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> springtime.music
'Kommisar - Springtime.mp3'
>>> springtime.samplestart
'105.760'
>>> springtime.labels
'0=Song Start'
```

Refer to *Known properties* for the full list of attributes for each simfile format. Many properties are shared between the SM and SSC formats, so you can use them without checking what kind of *Simfile* or *Chart* you have.

All properties return a string value, or None if the property is missing. The possibility of None can be annoying in type-checked code, so you may want to write expressions like `sf.title or ""` to guarantee a string.

Attributes are great, but they can't cover *every* property found in every simfile in existence. When you need to deal with unknown properties, you can use any simfile or chart as a dictionary of uppercase property names (they all extend `OrderedDict` under the hood):

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> springtime['ARTIST']
'Kommisar'
>>> springtime['ARTIST'] is springtime.artist
True
>>> for property, value in springtime.items():
...     if property == 'TITLETRANSLIT': break
...     print(property, '=', repr(value))
...
VERSION = '0.83'
TITLE = 'Springtime'
SUBTITLE = ''
ARTIST = 'Kommisar'
```

Note: One consequence of the backing `OrderedDict` is that **duplicate properties are not preserved.** This is a rare occurrence among existing simfiles, usually indicative of manual editing, and it doesn't appear to have any practical use case. However, if the loss of this information is a concern, consider using msdparser to stream the key-value pairs directly.

3.2.3 Accessing charts

Charts are different from regular properties, because a simfile can have zero to many charts. The charts are stored in a list under the *charts* attribute:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> len(springtime.charts)
9
>>> springtime.charts[0]
<SSCChart: dance-single Challenge 12>
```

To find a particular chart, use a for-loop or Python's built-in `filter` function:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> list(filter(
...     lambda chart: chart.stepstype == 'pump-single' and int(chart.meter) > 20,
...     springtime.charts,
... ))
...
[<SSCChart: pump-single Challenge 21>]
```

Much like simfiles, charts have their own "known properties" like `meter` and `stepstype` which can be fetched via attributes, as well as a backing `OrderedDict` which maps uppercase keys like `'METER'` and `'STEPSTYPE'` to the same string values.

Warning: Even the *`meter`* property is a string! Some simfiles in the wild have a non-numeric meter due to manual editing; it's up to client code to determine how to deal with this.

If you need to compare meters numerically, you can use `int(chart.meter)`, or `int(chart.meter or '1')` to sate type-checkers like mypy.

3.2.4 Editing simfile data

Simfile and chart objects are mutable: you can add, change, and delete properties and charts through the usual Python mechanisms.

Changes to known properties are kept in sync between the attribute and key lookups; the attributes are Python properties that use the key lookup behind the scenes.

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> springtime.subtitle = '(edited)'
>>> springtime
<SSCSimfile: Springtime (edited)>
>>> springtime.charts.append(SMChart())
>>> len(springtime.charts)
10
>>> del springtime.displaybpm
>>> 'DISPLAYBPM' in springtime
False
```

If you want to change more complicated data structures like timing and note data, refer to *Timing & note data* for an overview of the available classes & functions, rather than operating on the string values directly.

```
>>> import simfile
>>> from simfile.notes import NoteData
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> first_chart = springtime.charts[0]
>>> notedata = NoteData(first_chart)
>>> # (...modify the note data...)
>>> first_chart.notes = str(notedata)
```

Note: The keys of an *`SMChart`* are static; they can't be added or removed, but their values can be replaced.

3.2.5 Writing simfiles to disk

There are a few options for saving simfiles to the filesystem. If you want to read simfiles from the disk, modify them, and then save them, you can use the *simfile.mutate()* context manager:

```
>>> import simfile
>>> input_filename = 'testdata/Springtime/Springtime.ssc'
>>> with simfile.mutate(
...     input_filename,
...     backup_filename=f'{input_filename}.old',
... ) as springtime:
...     if springtime.subtitle.endswith('(edited)'):
...         raise simfile.CancelMutation
...     springtime.subtitle += '(edited)'
```

In this example, we specify the optional *backup_filename* parameter to preserve the simfile's original contents. Alternatively, we could have specified an *output_filename* to write the modified simfile somewhere other than the input filename.

simfile.mutate() writes the simfile back to the disk only if it exits without an exception. Any exception that reaches the context manager will propagate up, *except* for *CancelMutation*, which cancels the operation without re-throwing.

If this workflow doesn't suit your use case, you can serialize to a file object using the simfile's *serialize()* method:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> springtime.subtitle = '(edited)'
>>> with open('testdata/Springtime (edit).ssc', 'w', encoding='utf-8') as outfile:
...     springtime.serialize(outfile)
```

Finally, if your destination isn't a file object, you can serialize the simfile to a string using `str(simfile)` and proceed from there.

3.2.6 Robust parsing of arbitrary simfiles

The real world is messy, and many simfiles on the Internet are technically malformed despite appearing to function correctly in StepMania. This library aims to be **strict by default**, both for input and output, but allow more permissive input handling on an opt-in basis.

The functions exposed by the top-level *simfile* module accept a *strict* parameter that can be set to False to suppress MSD parser errors:

```
>>> import simfile
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc', strict=False)
```

> **Warning:** Due to the simplicity of the MSD format, there's only one error condition at the data layer - stray text between parameters - which setting *strict* to False suppresses. Almost any text file will successfully parse as a "simfile" with this check disabled, so exercise caution when applying this feature to arbitrary files.

While most modern simfiles are encoded in UTF-8, many older simfiles use dated encodings (perhaps resembling Latin-1 or Shift-JIS). This was a pain to handle correctly in older versions, but in version 2.0, all *simfile* functions that interact with the filesystem detect an appropriate encoding automatically, so there's typically no need to specify an encoding or handle `UnicodeDecodeError` exceptions. Read through the documentation of *open_with_detected_encoding()* for more details.

When grouping notes using the `group_notes()` function, orphaned head or tail notes will raise an exception by default. Refer to *Handling holds, rolls, and jumps* for more information on handling orphaned notes gracefully. (This is more common than you might imagine - "Springtime", which comes bunded with StepMania, has orphaned tail notes in its first chart!)

3.3 Known properties

Known properties refer to properties of simfiles and their charts that the current stable version of StepMania actively uses. These are the properties that **simfile** exposes as attributes on simfile and chart objects. They are also the only properties that StepMania's built-in editor will preserve when saving a simfile; unknown properties are liable to be deleted if saved by the StepMania editor.

When working with known properties, you should prefer to use attributes (e.g. `sim.title`, `chart.stepstype`) rather than indexing into the underlying dictionary (e.g. `sim['TITLE']`, `chart['STEPSTYPE']`). While these are functionally equivalent in many cases, attributes generally behave closer to how StepMania interprets simfiles:

- If a property is missing from the simfile, accessing the attribute returns None instead of throwing a `KeyError`. StepMania generally treats missing properties as if they had an empty or default value, so it's nice to be able to handle this case without having to catch exceptions everywhere.
- StepMania supports a few legacy aliases for properties, and attributes make use of these aliases when present. For example, if a simfile contains a `FREEZES` property instead of the usual `STOPS`, `sim.stops` will use the alias in the backing dictionary (both for reads and writes!), whereas `sim['STOPS']` will throw a `KeyError`. This lets you write cleaner code with fewer special cases for old simfiles.
- Attributes are implicitly spell-checked: misspelling a property like `sim.artistranslit` will consistently raise an `AttributeError`, and may even be flagged by your IDE depending on its Python type-checking capabilities. By contrast, reading from `sim['ARTISTRANSLIT']` will generally raise the more vague `KeyError` exception, and writing to such a field would create a new, unknown property in the simfile, which is probably not what you wanted. Furthermore, your IDE would have no way to know the string property is misspelled.

With that said, there are legitimate use cases for indexing. String keys are easier when you need to operate on multiple properties generically, and they're the only option for accessing "unknown properties" like numbered `BGCHANGES` and properties only used by derivatives of StepMania. When dealing with property string keys, consider using the `.get` method from the underlying dictionary to handle missing keys gracefully.

3.3.1 What are the known properties?

These are the known properties for simfiles:

String key	Attribute	SMSimfile	SSCSimfile
TITLE	title	✓	✓
SUBTITLE	subtitle	✓	✓
ARTIST	artist	✓	✓
TITLETRANSLIT	titletranslit	✓	✓
SUBTITLETRANSLIT	subtitletranslit	✓	✓
ARTISTTRANSLIT	artisttranslit	✓	✓
GENRE	genre	✓	✓
CREDIT	credit	✓	✓
BANNER	banner	✓	✓
BACKGROUND	background	✓	✓

continues on next page

Table 1 – continued from previous page

String key	Attribute	SMSimfile	SSCSimfile
LYRICSPATH	lyricspath	✓	✓
CDTITLE	cdtitle	✓	✓
MUSIC	music	✓	✓
OFFSET	offset	✓	✓
BPMS	bpms	✓	✓
STOPS	stops	✓	✓
FREEZES[1]	stops	✓	
DELAYS	delays	✓	✓
TIMESIGNATURES	timesignatures	✓	✓
TICKCOUNTS	tickcounts	✓	✓
INSTRUMENTTRACK	instrumenttrack	✓	✓
SAMPLESTART	samplestart	✓	✓
SAMPLELENGTH	samplelength	✓	✓
DISPLAYBPM	displaybpm	✓	✓
SELECTABLE	selectable	✓	✓
BGCHANGES	bgchanges	✓	✓
ANIMATIONS[1]	bgchanges	✓	✓
FGCHANGES	fgchanges	✓	✓
KEYSOUNDS	keysounds	✓	✓
ATTACKS	attacks	✓	✓
VERSION	version		✓
ORIGIN	origin		✓
PREVIEWVID	previewvid		✓
JACKET	jacket		✓
CDIMAGE	cdimage		✓
DISCIMAGE	discimage		✓
PREVIEW	preview		✓
MUSICLENGTH	musiclength		✓
LASTSECONDHINT	lastsecondhint		✓
WARPS	warps		✓
LABELS	labels		✓
COMBOS	combos		✓
SPEEDS	speeds		✓
SCROLLS	scrolls		✓
FAKES	fakes		✓

And these are the known properties for charts:

[1] These keys are aliases supported by StepMania. The attribute will only use the alias if it's present in the backing dictionary and the standard name is not.

String key	Attribute	SMChart	SSCChart
STEPSTYPE	stepstype	✓	✓
DESCRIPTION	description	✓	✓
DIFFICULTY	difficulty	✓	✓
METER	meter	✓	✓
RADARVALUES	radarvalues	✓	✓
NOTES	notes	✓	✓
NOTES2[1]	notes		✓
CHARTNAME	chartname		✓
CHARTSTYLE	chartstyle		✓
CREDIT	credit		✓
MUSIC	music		✓
BPMS	bpms		✓
STOPS	stops		✓
DELAYS	delays		✓
TIMESIGNATURES	timesignatures		✓
TICKCOUNTS	tickcounts		✓
COMBOS	combos		✓
WARPS	warps		✓
SPEEDS	speeds		✓
SCROLLS	scrolls		✓
FAKES	fakes		✓
LABELS	labels		✓
ATTACKS	attacks		✓
OFFSET	offset		✓
DISPLAYBPM	displaybpm		✓

Known properties supported by both the SM and SSC formats are documented in `BaseSimfile` and `BaseChart`. These are exactly the known properties for `SMSimfile` and `SMChart`. The SSC format then adds additional known properties on top of the base set.

Here is all of the relevant documentation:

class simfile.base.**BaseSimfile**(*, *file: Optional[Union[TextIO, Iterator[str]]] = None*, *string: Optional[str] = None*, *strict: bool = True*)

A simfile, including its metadata (e.g. song title) and charts.

Metadata is stored directly on the simfile object through a dict-like interface. Keys are unique (if there are duplicates, the last value wins) and converted to uppercase.

Additionally, properties recognized by the current stable version of StepMania are exposed through lower-case properties on the object for easy (and implicitly spell-checked) access. The following known properties are defined:

- Metadata: *title*, *subtitle*, *artist*, *titletranslit*, *subtitletranslit*, *artisttranslit*, *genre*, *credit*, *samplestart*, *samplelength*, *selectable*, *instrumenttrack*, *timesignatures*
- File paths: *banner*, *background*, *lyricspath*, *cdtitle*, *music*
- Gameplay events: *bgchanges*, *fgchanges*, *keysounds*, *attacks*, *tickcounts*
- Timing data: *offset*, *bpms*, *stops*, *delays*

If a desired simfile property isn't in this list, it can still be accessed as a dict item.

By default, the underlying parser will throw an exception if it finds any stray text between parameters. This behavior can be overridden by setting *strict* to False in the constructor.

class simfile.base.**BaseChart**
One chart from a simfile.

All charts have the following known properties: *stepstype*, *description*, *difficulty*, *meter*, *radarvalues*, and *notes*.

class simfile.ssc.**SSCSimfile**(*, *file: Optional[Union[TextIO, Iterator[str]]] = None*, *string: Optional[str] = None*, *strict: bool = True*)
SSC implementation of *BaseSimfile*.

Adds the following known properties:

- SSC version: *version*
- Metadata: *origin*, *labels*, *musiclength*, *lastsecondhint*
- File paths: *previewvid*, *jacket*, *cdimage*, *discimage*, *preview*
- Gameplay events: *combos*, *speeds*, *scrolls*, *fakes*
- Timing data: *warps*

class simfile.ssc.**SSCChart**
SSC implementation of *BaseChart*.

Unlike *SMChart*, SSC chart metadata is stored as key-value pairs, so this class allows full modification of its backing OrderedDict.

Adds the following known properties:

- Metadata: *chartname*, *chartstyle*, *credit*, *timesignatures*
- File paths: *music*
- Gameplay events: *tickcounts*, *combos*, *speeds*, *scrolls*, *fakes*, *attacks*
- Timing data: *bpms*, *stops*, *delays*, *warps*, *labels*, *offset*, *displaybpm*

3.4 Timing & note data

For advanced use cases that require parsing specific fields from simfiles and charts, **simfile** provides subpackages that interface with the core simfile & chart classes. As of version 2.0, the available subpackages are *simfile.notes* and *simfile.timing*.

3.4.1 Reading note data

The primary function of a simfile is to store charts, and the primary function of a chart is to store *note data* – sequences of inputs that the player must follow to the rhythm of the song. Notes come in different types, appear in different columns, and occur on different beats of the chart.

Rather than trying to parse a chart's *notes* field directly, use the *NoteData* class:

```
>>> import simfile
>>> from simfile.notes import NoteData
>>> from simfile.timing import Beat
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> chart = springtime.charts[0]
>>> note_data = NoteData(chart)
>>> note_data.columns
4
>>> for note in note_data:
```

(continues on next page)

(continued from previous page)

```
...     if note.beat > Beat(18): break
...     print(note)
...
Note(beat=Beat(16), column=2, note_type=NoteType.TAP)
Note(beat=Beat(16.5), column=2, note_type=NoteType.TAP)
Note(beat=Beat(16.75), column=0, note_type=NoteType.TAP)
Note(beat=Beat(17), column=1, note_type=NoteType.TAP)
Note(beat=Beat(17.5), column=0, note_type=NoteType.TAP)
Note(beat=Beat(17.75), column=2, note_type=NoteType.TAP)
Note(beat=Beat(18), column=1, note_type=NoteType.TAP)
```

There's no limit to how many notes a chart can contain – some have tens or even hundreds of thousands! For this reason, *NoteData* only generates *Note* objects when you ask for them, one at a time, rather than storing a list of notes. Likewise, functions in this library that operate on note data accept an iterator of notes, holding them in memory for as little time as possible.

3.4.2 Counting notes

Counting notes isn't as straightforward as it sounds: there are different note types and different ways to handle notes on the same beat. StepMania offers six different "counts" on the music selection screen by default, each offering a unique aggregation of the gameplay events in the chart.

To reproduce StepMania's built-in note counts, use the functions provided by the *simfile.notes.count* module:

```
>>> import simfile
>>> from simfile.notes import NoteData
>>> from simfile.notes.count import *
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> chart = springtime.charts[0]
>>> note_data = NoteData(chart)
>>> count_steps(note_data)
864
>>> count_jumps(note_data)
33
```

If the functions in this module aren't sufficient for your needs, move on to the next section for more options.

3.4.3 Handling holds, rolls, and jumps

Conceptually, **hold** and **roll** notes are atomic: while they have discrete start and end beats, *both* endpoints must be specified for the note to be valid. This logic also extends to **jumps** in certain situations: for example, combo counters, judgement & score algorithms, and note counting methods may consider jumps to be "equal" in some sense to isolated tap notes.

In contrast, iterating over *NoteData* yields a separate "note" for every discrete event in the chart: hold and roll heads are separate from their tails, and jumps are emitted one note at a time. You may want to group either or both of these types of notes together, depending on your use case.

The *group_notes()* function handles all of these cases. In this example, we find that the longest hold in Springtime's Lv. 21 chart is 6½ beats long:

```
>>> import simfile
>>> from simfile.notes import NoteType, NoteData
>>> from simfile.notes.group import OrphanedNotes, group_notes
```

(continues on next page)

(continued from previous page)

```
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> chart = next(filter(lambda chart: chart.meter == '21', springtime.charts))
>>> note_data = NoteData(chart)
>>> group_iterator = group_notes(
...     note_data,
...     include_note_types={NoteType.HOLD_HEAD, NoteType.TAIL},
...     join_heads_to_tails=True,
...     orphaned_tail=OrphanedNotes.DROP_ORPHAN,
... )
>>> longest_hold = 0
>>> for grouped_notes in group_iterator:
...     note = note_group[0]
...     longest_hold = max(longest_hold, note.tail_beat - note.beat)
...
>>> longest_hold
Fraction(13, 2)
```

There's a lot going on in this code snippet, so here's a breakdown of the important parts:

```
>>> group_iterator = group_notes(
...     note_data,
...     include_note_types={NoteType.HOLD_HEAD, NoteType.TAIL},
...     orphaned_tail=OrphanedNotes.DROP_ORPHAN,
...     join_heads_to_tails=True,
... )
```

Here we choose to group hold heads to their tails, dropping any orphaned tails. By default, orphaned heads or tails will raise an exception, but in this example we've opted out of including roll heads, whose tails would become orphaned. If we chose to include *NoteType.ROLL_HEAD* in the set, then we could safely omit the *orphaned_tail* argument since all tails should have a matching head (assuming the chart is valid).

```
>>> for grouped_notes in group_iterator:
...     note = note_group[0]
...     longest_hold = max(longest_hold, note.tail_beat - note.beat)
```

The *group_notes()* function yields *lists of notes* rather than single notes. In this example, every list will only have a single element because we haven't opted into joining notes that occur on the same beat (we would do so using the *same_beat_notes* parameter). As such, we can extract the single note by indexing into each note group.

You'll notice that we're using a *tail_beat* attribute, which isn't present in the *Note* class. That's because these notes are actually *NoteWithTail* instances: the *lists of notes* referenced above are actually lists of *Note* and/or *NoteWithTail* objects, depending on the parameters. In this case, we know that *every* note will be a *NoteWithTail* instance because we've only included head and tail note types, which will be joined together.

Out of all the possible combinations of *group_notes()* parameters, this example yields fairly simple items (singleton lists of *NoteWithTail* instances). Other combinations of parameters may yield variable-length lists where you need to explicitly check the type of the elements.

3.4.4 Changing & writing notes

As mentioned before, the *simfile.notes* API operates on *iterators* of notes to keep the memory footprint light. Iterating over *NoteData* is one way to obtain a note iterator, but you can also generate *Note* objects yourself.

To serialize a stream of notes into note data, use the class method *NoteData.from_notes()*:

```
>>> import simfile
>>> from simfile.notes import Note, NoteType, NoteData
>>> from simfile.timing import Beat
>>> cols = 4
>>> notes = [
...     Note(beat=Beat(i, 2), column=i%cols, note_type=NoteType.TAP)
...     for i in range(8)
... ]
>>> note_data = NoteData.from_notes(notes, cols)
>>> print(str(note_data))
1000
0100
0010
0001
1000
0100
0010
0001
```

The `notes` variable above *could* use parentheses to define a generator instead of square brackets to define a list, but you don't have to stick to pure generators to interact with the *simfile.notes* API. **Use whatever data structure suits your use case,** as long as you're cognizant of potential out-of-memory conditions.

> **Warning:** Note iterators passed to the *simfile.notes* API should always be sorted by their natural ordering, the same order in which they appear in strings of note data (and the order you'll get by iterating over *NoteData*). If necessary, you can use Python's built-in sorting mechanisms on *Note* objects to ensure they are in the right order, like `sorted()`, `list.sort()`, and the `bisect` module.

To insert note data back into a chart, convert it to a string and assign it to the chart's *notes* attribute. In this example, we mirror the notes' columns in Springtime's first chart and update the simfile object:

```
>>> import simfile
>>> from simfile.notes import NoteData
>>> from simfile.notes.count import *
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> chart = springtime.charts[0]
>>> note_data = NoteData(chart)
>>> cols = note_data.columns
>>> def mirror(note, cols):
...     return Note(
...         beat=note.beat,
...         column=cols - note.column - 1,
...         note_type=note.note_type,
...     )
...
>>> mirrored_notes = (mirror(note, cols) for note in note_data)
>>> mirrored_note_data = NoteData.from_notes(mirrored_notes, cols)
>>> chart.notes = str(mirrored_note_data)
```

From there, we could write the modified simfile back to disk as described in *Reading & writing simfiles*.

3.4.5 Reading timing data

Rather than reading fields like `BPMS` and `STOPS` directly from the simfile, use the *`TimingData`* class:

```
>>> import simfile
>>> from simfile.timing import TimingData
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> timing_data = TimingData(springtime)
>>> timing_data.bpms
BeatValues([BeatValue(beat=Beat(0), value=Decimal('181.685'))])
```

The SSC format introduces "split timing" – per-chart timing data – which *`TimingData`* empowers you to handle as effortlessly as providing the chart:

```
>>> import simfile
>>> from simfile.timing import TimingData
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> chart = springtime.charts[0]
>>> split_timing = TimingData(springtime, chart)
>>> split_timing.bpms
BeatValues([BeatValue(beat=Beat(0), value=Decimal('181.685')),␣
↪BeatValue(beat=Beat(304), value=Decimal('90.843')), BeatValue(beat=Beat(311),␣
↪value=Decimal('181.685'))])
```

This works regardless of whether the chart has split timing, or even whether the simfile is an SSC file; if the chart has no timing data of its own, it will be ignored and the simfile's timing data will be used instead.

3.4.6 Getting the displayed BPM

On StepMania's music selection screen, players can typically see the selected chart's BPM, whether static or a range of values. For most charts, this is inferred through its timing data, but the *DISPLAYBPM* tag can be used to override this value. Additionally, the special *DISPLAYBPM* value `*` obfuscates the BPM on the song selection screen, typically with a flashing sequence of random numbers.

To get the displayed BPM, use the *`displaybpm()`* function:

```
>>> import simfile
>>> from simfile.timing.displaybpm import displaybpm
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> disp = displaybpm(springtime)
>>> if disp.value:
...     print(f"Static value: {disp.value}")
... elif disp.range:
...     print(f"Range of values: {disp.value[0]}-{disp.value[1]}")
... else:
...     print(f"* (obfuscated BPM)")
...
Static value: 182
```

The return value will be one of *`StaticDisplayBPM`*, *`RangeDisplayBPM`*, or *`RandomDisplayBPM`*. All of these classes implement four properties *(as of 2.1)*:

- *`StaticDisplayBPM.value`* returns the single BPM value; the other classes return None.
- *`RangeDisplayBPM.range`* returns a (min, max) tuple; the other classes return None.

- *min* and *max* return the lowest and highest BPM values for both *StaticDisplayBPM* and *RangeDisplayBPM* (they will be equal for the static case).

Here's the same information in a table:

Actual BPM	*DISPLAYBPM* value	Class	min	max	value	range
300		StaticDisplayBPM	300	300	300	None
12-300	300	StaticDisplayBPM	300	300	300	None
12-300		RangeDisplayBPM	12	300	None	(12, 300)
12-300	150:300	RangeDisplayBPM	150	300	None	(150, 300)
12-300	*	RandomDisplayBPM	None	None	None	None

Much like *TimingData*, *displaybpm()* accepts an optional chart parameter for SSC split timing.

Also, setting *ignore_specified* to True will ignore any *DISPLAYBPM* property and always return the true BPM range. If you want to get the real BPM hidden by RandomDisplayBPM (while allowing numeric *DISPLAYBPM* values), you can do something like this:

```
disp = displaybpm(sf)
if not disp.max: # RandomDisplayBPM case
    disp = displaybpm(sf, ignore_specified=True)
```

Warning: It may be tempting to use *max* to calculate the scroll rate for MMod, but this will be incorrect in some edge cases, most notably songs with very high BPMs and no *DISPLAYBPM* specified.

3.4.7 Converting song time to beats

If you wanted to implement a simfile editor or gameplay engine, you'd need some way to convert song time to beats and vice-versa. To reach feature parity with StepMania, you'd need to implement BPM changes, stops, delays, and warps in order for your application to support all the simfiles that StepMania accepts.

Consider using the *TimingEngine* for this use case:

```
>>> import simfile
>>> from simfile.timing import Beat, TimingData
>>> from simfile.timing.engine import TimingEngine
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> timing_data = TimingData(springtime)
>>> engine = TimingEngine(timing_data)
>>> engine.time_at(Beat(32))
10.658
>>> engine.beat_at(10.658)
Beat(32)
```

This engine handles all of the timing events described above, including edge cases involving overlapping stops, delays, and warps. You can even check whether a note near a warp segment would be *hittable()* or not!

3.4.8 Combining notes and time

Finally, to tie everything together, check out the *time_notes()* function which converts a *Note* stream into a *TimedNote* stream:

```
>>> import simfile
>>> from simfile.timing import Beat, TimingData
>>> from simfile.notes import NoteData
>>> from simfile.notes.timed import time_notes
>>> springtime = simfile.open('testdata/Springtime/Springtime.ssc')
>>> chart = springtime.charts[0]
>>> note_data = NoteData(chart)
>>> timing_data = TimingData(springtime, chart)
>>> for timed_note in time_notes(note_data, timing_data):
...     if 60 < timed_note.time < 61:
...         print(timed_note)
...
TimedNote(time=60.029, note=Note(beat=Beat(181.5), column=3, note_type=NoteType.TAP))
TimedNote(time=60.194, note=Note(beat=Beat(182), column=0, note_type=NoteType.HOLD_
↪HEAD))
TimedNote(time=60.524, note=Note(beat=Beat(183), column=3, note_type=NoteType.TAP))
TimedNote(time=60.855, note=Note(beat=Beat(184), column=2, note_type=NoteType.TAP))
```

You could use this to determine the notes per second (NPS) over the entire chart, or at a specific time like the example above. Get creative!

3.5 Learn by example

This page includes examples of varying length demonstrating correct & type-checked **simfile** library usage. You're free to use these recipes & scripts as-is, modify them to suit your needs, or simply use them as a learning aid.

3.5.1 Recipes

This section includes short snippets of code that demonstrate basic library usage. These recipes are in the public domain.

Get charts for one game mode

```
from typing import Iterator
from simfile.types import Chart

# Imperative version
def charts_for_stepstype(charts, stepstype='dance-single') -> Iterator[Chart]:
    for chart in charts:
        if chart.stepstype == stepstype:
            yield chart

# One-liner version
def charts_for_stepstype(charts, stepstype='dance-single') -> Iterator[Chart]:
    yield from filter(lambda chart: chart.stepstype == stepstype, charts)
```

Get the hardest chart

```
from typing import Optional, Sequence
from simfile.types import Chart

# Imperative version
def get_hardest_chart(charts) -> Optional[Chart]:
    hardest_chart: Optional[Chart] = None
    hardest_meter: Optional[int] = None

    for chart in charts:
        # Remember to convert `meter` to an integer for comparisons
        meter = int(chart.meter or "1")
        if hardest_meter is None or meter > hardest_meter:
            hardest_chart = chart
            hardest_meter = meter

    return hardest_chart

# One-liner version
def get_hardest_chart(charts: Sequence[Chart]) -> Optional[Chart]:
    return max(
        charts,
        key=lambda chart: int(chart.meter or "1"),
        default=None,
    )
```

Mirror a chart's notes

```
from typing import Iterator
from simfile.types import Chart
from simfile.notes import Note, NoteData

def mirror_note(note: Note, columns: int) -> Note:
    # Make a new Note with all fields the same except for column
    return note._replace(
        # You could replace this expression with anything you want
        column=columns - note.column - 1
    )

def mirror_notes(notedata: NoteData) -> Iterator[Note]:
    columns = notedata.columns
    for note in notedata:
        yield mirror_note(note, columns)

def mirror_chart_in_place(chart: Chart) -> None:
    notedata = NoteData(chart)
    mirrored = NoteData.from_notes(
        mirror_notes(notedata),
        columns=notedata.columns,
    )
    # Assign str(NoteData) to Chart.notes to update the chart's notes
    chart.notes = str(mirrored)
```

Remove all but one chart from a simfile

```
from typing import Optional
from simfile.types import Chart, Charts, Simfile

# When you have multiple parameters of the same type (str in this case),
# it's good practice to use a * pseudo-argument to require them to be named
def find_chart(charts: Charts, *, stepstype: str, difficulty: str) -> Optional[Chart]:
    for chart in charts:
        if chart.stepstype == stepstype and chart.difficulty == difficulty:
            return chart

def remove_other_charts(sf: Simfile, *, stepstype='dance-single', difficulty=
↪'Challenge'):
    the_chart = find_chart(sf.charts, stepstype=stepstype, difficulty=difficulty)
    if the_chart:
        # Replace the simfile's charts with a list of one
        sf.charts = [the_chart]  # type: ignore
    else:
        # You could alternatively raise an exception, pick a different chart,
        # set sf.charts to an empty list, etc.
        print(f"No {stepstype} {difficulty} chart found for {repr(sf)}")
```

3.5.2 Full scripts

This section includes complete, ready-to-use scripts that automate repetitive tasks on simfile packs. These scripts are licensed under the MIT License, the same license as the **simfile** library itself.

change_sync_bias.py

```
R"""
Add or subtract the standard ITG sync bias (9 milliseconds)
to all of the sync offsets in a pack.

This script updates the offsets of both SM and SSC simfiles,
including any SSC charts with their own timing data.

If you actually intend to use this script in practice,
you may want to keep track of which packs you've already adjusted
using a text file in each pack directory or some other system.

Usage examples:

    # Convert a pack from "null sync" to "ITG sync"
    python change_sync_bias.py +9 "C:\StepMania\Songs\My Pack"

    # Convert a pack from "ITG sync" to "null sync"
    python change_sync_bias.py -9 "C:\StepMania\Songs\My Pack"
"""
import argparse
from decimal import Decimal
import sys
from typing import Union
```

(continues on next page)

(continued from previous page)

```
import simfile
import simfile.dir


class ChangeSyncBiasArgs:
    """Stores the command-line arguments for this script."""

    pack: str
    itg_to_null: bool
    null_to_itg: bool


def argparser():
    """Get an ArgumentParser instance for this command-line script."""
    parser = argparse.ArgumentParser(prefix_chars="-+")
    parser.add_argument("pack", type=str, help="path to the pack to modify")
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument(
        "-9", "--itg-to-null", action="store_true", help="subtract 9ms from offsets"
    )
    group.add_argument(
        "+9", "--null-to-itg", action="store_true", help="add 9ms to offsets"
    )
    return parser


def adjust_offset(
    obj: Union[simfile.types.Simfile, simfile.ssc.SSCChart],
    delta: Decimal,
):
    """Add the delta to the simfile or SSC chart's offset, if present."""
    if obj.offset is not None:
        obj.offset = str(Decimal(obj.offset) + delta)


def change_sync_bias(simfile_path: str, args: ChangeSyncBiasArgs):
    """
    Add or subtract 9 milliseconds to the simfile's offset,
    as well as any SSC charts with their own timing data.

    This saves the updated simfile to its original location
    and writes a backup copy with a ~ appended to the filename.
    """
    # Map the +9 or -9 arg to the actual offset delta.
    #
    # We don't have to check both itg_to_null and null_to_itg
    # because the mutually exclusive & required argument group
    # ensures that exactly one of them will be True.
    delta = Decimal("-0.009" if args.itg_to_null else "+0.009")

    # You could specify output_filename here to write the updated file elsewhere
    with simfile.mutate(
        input_filename=f"{simfile_path}",
        backup_filename=f"{simfile_path}~",
    ) as sf:
        print(f"Processing {simfile_path}")
```

(continues on next page)

(continued from previous page)

```
        # Always adjust the simfile's offset
        adjust_offset(sf, delta)

        # Additionally try to adjust SSC charts' offsets.
        # This won't do anything unless the chart has its own timing data.
        if isinstance(sf, simfile.ssc.SSCSimfile):
            for chart in sf.charts:
                adjust_offset(chart, delta)


def main(argv):
    # Parse command-line arguments
    args = argparser().parse_args(argv[1:], namespace=ChangeSyncBiasArgs())

    # Iterate over SimfileDirectory objects from the pack
    # so that we can easily get the .sm and/or .ssc paths
    for simfile_dir in simfile.dir.SimfilePack(args.pack).simfile_dirs():

        # Try to update whichever formats exist
        for simfile_path in [simfile_dir.sm_path, simfile_dir.ssc_path]:
            if simfile_path:
                change_sync_bias(simfile_path, args)


if __name__ == "__main__":
    main(sys.argv)
```

sort_by_difficulty.py

```
R"""
Change the title of every simfile in a pack
so that they are sorted by difficulty in StepMania.

This script finds the hardest chart of a given stepstype (dance-single by default)
and puts its meter (difficulty number) between brackets at the start of the title
and titletranslit.

Usage examples:

    # Sort a pack by difficulty
    python sort_by_difficulty.py "C:\StepMania\Songs\My Pack"

    # Unsort by difficulty (remove the title prefixes)
    python sort_by_difficulty.py -r "C:\StepMania\Songs\My Pack"

    # Customize stepstype and digits
    python sort_by_difficulty.py -s dance-double -d 3 "C:\StepMania\My Pack"
"""
import argparse
import sys
from typing import Optional, Sequence

import simfile
import simfile.dir
```

(continues on next page)

(continued from previous page)

```
class SortByDifficultyArgs:
    """Stores the command-line arguments for this script."""

    pack: str
    stepstype: str
    digits: int
    remove: bool


def argparser():
    """Get an ArgumentParser instance for this command-line script."""
    parser = argparse.ArgumentParser()
    parser.add_argument("pack", type=str, help="path to the pack to modify")
    parser.add_argument("-s", "--stepstype", type=str, default="dance-single")
    parser.add_argument(
        "-d",
        "--digits",
        type=int,
        default=2,
        help="minimum digits (will add leading zeroes)",
    )
    parser.add_argument(
        "-r",
        "--remove",
        action=argparse.BooleanOptionalAction,
        help="remove meter prefix",
    )
    return parser


def hardest_chart(
    charts: Sequence[simfile.types.Chart], stepstype: str
) -> Optional[simfile.types.Chart]:
    """
    Find & return the hardest chart (numerically) of a given stepstype.

    Returns None if there are no charts matching the stepstype.
    """
    return max(
        [c for c in charts if c.stepstype == stepstype],
        key=lambda c: int(c.meter or "1"),
        default=None,
    )


def prefix_title_with_meter(simfile_path: str, args: SortByDifficultyArgs):
    """
    Add (or remove) a numeric prefix to the simfile's title and titletranslit.

    This saves the updated simfile to its original location
    and writes a backup copy with a ~ appended to the filename.
    """
    # You could specify output_filename here to write the updated file elsewhere
    with simfile.mutate(
        input_filename=f"{simfile_path}",
        backup_filename=f"{simfile_path}~",
```

(continues on next page)

(continued from previous page)

```
    ) as sf:
        print(f"Processing {simfile_path}")

        # It's very unlikely for the title property to be blank or missing.
        # This is mostly to satisfy type-checkers.
        current_title = sf.title or ""
        current_titletranslit = sf.titletranslit or ""

        if args.remove:
            def remove_starting_brackets(current_text: str) -> str:
                """
                If current_text has a bracketed number at the start of the text,␣
↪remove it and return it
                Otherwise, return current_text unchanged.
                """
                # Look for a number in brackets at the start of the text
                if current_text.startswith("["):
                    open_bracket_index = current_text.find("[")
                    close_bracket_index = current_text.find("]")
                    bracketed_text = current_text[
                        open_bracket_index + 1 : close_bracket_index
                    ]
                    if bracketed_text.isnumeric():
                        # Remove the bracketed number from the text
                        return current_title[close_bracket_index + 1 :].lstrip(" ")
                return current_title
            sf.title = remove_starting_brackets(sf.title)
            sf.titletranslit = remove_starting_brackets(sf.titletranslit)
        else:
            # Find the hardest chart (numerically) within a stepstype
            # and use it to prefix the title
            chart = hardest_chart(sf.charts, args.stepstype)

            # Skip this simfile if there were no charts for the stepstype.
            # Nothing will be written to disk in this case.
            if not chart:
                raise simfile.CancelMutation

            # It's very unlikely for the meter property to be blank or missing.
            # This is mostly to satisfy type-checkers.
            meter = chart.meter or "1"

            # Put the meter at the start of the title,
            # filling in leading zeros per arguments
            sf.title = f"[{meter.zfill(args.digits)}] {current_title}"
            sf.titletranslit = f"[{meter.zfill(args.digits)}] {current_titletranslit}"


def main(argv):
    # Parse command-line arguments
    args = argparser().parse_args(argv[1:], namespace=SortByDifficultyArgs())

    # Iterate over SimfileDirectory objects from the pack
    # so that we can easily get the .sm and/or .ssc paths
    for simfile_dir in simfile.dir.SimfilePack(args.pack).simfile_dirs():

        # Try to update whichever formats exist
```

(continues on next page)

(continued from previous page)

```
        for simfile_path in [simfile_dir.sm_path, simfile_dir.ssc_path]:
            if simfile_path:
                prefix_title_with_meter(simfile_path, args)


if __name__ == "__main__":
    main(sys.argv)
```

3.6 Changelog

3.6.1 2.1.1

Bugfixes

Two bugs in **simfile** 2.1.0's SSC implementation broke multi-value properties, causing them to be truncated or mangled past the first value. This release fixes these issues:

1. When opening an SSC file, the *DISPLAYBPM* and *ATTACKS* properties of both simfiles and charts no longer stop parsing at the first `:`. For *DISPLAYBPM*, this meant a BPM range of `120:240` would have been incorrectly parsed as a static BPM of `120`. *ATTACKS* were completely broken as they use colon as a separator.
2. The aforementioned properties are now correctly serialized from *SSCChart*; previously, they would have been escaped with backslashes. This bug had the same effects described above, but only affected manual assignment of multi-value properties (e.g. `chart.displaybpm = "120:240"`) since the first bug shadowed this bug during deserialization.

3.6.2 2.1.0

New features

- The new *simfile.dir* module offers *SimfileDirectory* and *SimfilePack* classes for nagivating simfile filesystem structures.
- The new *simfile.assets* module provides an *Assets* class that can reliably discover paths to simfile assets, even if they're not specified in the simfile.
- The top-level *simfile* module now offers *opendir()* and *openpack()* functions as simplified interfaces to the *simfile.dir* API.
- PyFilesystem2 has been integrated throughout this library's filesystem interactions, enabling OS and non-OS filesystems to be traversed using the same code. All functions, methods, and constructors that lead to filesystem interactions now have an optional *filesystem* parameter for specifying a PyFS filesystem object. When omitted, the filesystem defaults to the native OS filesystem as before.
- The *DisplayBPM* classes now all expose the same four properties; the ones that don't apply to a particular class return None. This enables you to handle all three cases without having to import the types for `isinstance` checks. Refer to *Getting the displayed BPM* for more details.

Bugfixes

- The `charts` property on simfiles is now writable, meaning the list of charts can be overwritten directly (not just added to / removed from).
- Backslash escape sequences and multi-value MSD parameters are now handled correctly, both when opening and serializing simfiles. See the Enhancements section below for more details.
- *`sm_to_ssc()`* no longer produces invalid output when there are negative BPMs or stops in the timing data. (It throws `NotImplementedError` as a temporary stopgap. In the future, negative timing data will be converted to warps, as StepMania does automatically.)
- Various type annotations have been improved throughout the library. In particular, `Iterator` input arguments have been replaced with `Iterable` so that you don't need to wrap them in `iter(...)` to suppress type errors from static analyzers.

Enhancements

- The dependency on msdparser has been upgraded to version 2. This corrects parsing of escape sequences and multi-value parameters, meaning that `:` and `\` characters inside a value are handled the same way as in StepMania. Additionally, parsing is now up to 10 times faster than before!

3.6.3 2.0.1

Bugfix: The dependency on msdparser 1.0.0 was mis-specified in both the Pipfile and setup.py. Publishing msdparser 2.0.0-beta.3 (a breaking release) caused fresh installs to be broken. This patch fixes the version specification in both files.

3.6.4 2.0.0

Initial stable release of version 2. Refer to *Migrating from simfile 1.0 to 2.0* for a general overview of the changes since version 1.

3.7 Migrating from simfile 1.0 to 2.0

Version 1.0 of the **simfile** library was released in 2013. It only supported SM files and was primarily developed for Python 2, with support for Python 3 on a separate branch.

Version 2.0 is a near-complete rewrite of the library exclusively for Python 3, with SSC support as the flagship feature. Aside from new features, the design of the library has changed significantly to bring it in line with similar modern Python libraries.

3.7.1 Simfile & chart classes

In 1.0, the simfile & chart classes were `simfile.Simfile` and `simfile.Chart`.

In 2.0, the simfile & chart classes are split by simfile type:

- For SM files, the *simfile.sm* module provides *SMSimfile* and *SMChart*.
- For SSC files, the *simfile.ssc* module provides *SSCSimfile* and *SSCChart*.

Additionally, *simfile.types* provides the union types *Simfile* and *Chart*, which are used to annotate parameters & return types where either implementation is acceptable.

3.7.2 Reading simfiles

In 1.0, the `Simfile` constructor accepted a filename or file object, and a `.from_string` class method handled loading from string data:

```
>>> from simfile import Simfile # 1.0
>>> from_filename = Simfile('testdata/nekonabe/nekonabe.sm')
>>> # or...
>>> with open('testdata/nekonabe/nekonabe.sm', 'r') as infile:
...     from_file_obj = Simfile(infile)
...
>>> # or...
>>> from_string = Simfile.from_string(str(from_file_obj))
```

In 2.0, each of these options has a corresponding function in the top-level *simfile* module:

```
>>> import simfile # 2.0
>>> from_filename = simfile.open('testdata/nekonabe/nekonabe.sm')
>>> # or...
>>> with open('testdata/nekonabe/nekonabe.sm', 'r') as infile:
...     from_file_obj = simfile.load(infile)
...
>>> # or...
>>> from_string = simfile.loads(str(from_file_obj))
```

These methods determine which simfile format to use automatically, but you can alternatively instantiate the simfile classes directly. They take a *named* `file` or `string` argument:

```
>>> from simfile.sm import SMSimfile # 2.0
>>> with open('testdata/nekonabe/nekonabe.sm', 'r') as infile:
...     from_file_obj = SMSimfile(file=infile)
...
>>> # or...
>>> from_string = SMSimfile(string=str(from_file_obj))
```

3.7.3 Writing simfiles

In 1.0, simfile objects had a `.save` method that took a *maybe-optional* filename parameter:

```
>>> from simfile import Simfile # 1.0
>>> from_filename = Simfile('testdata/nekonabe/nekonabe.sm')        # filename␣
↪supplied
>>> from_filename.save()                                          # no problem!
>>> from_string = Simfile.from_string(str(from_filename)) # no filename supplied
>>> try:
...     from_string.save()                                        # to where?
... except ValueError:
...     from_string.save('testdata/nekonabe/nekonabe.sm')           # much better
```

In 2.0, simfile objects no longer know their own filenames. Either pass a file object to the simfile's *`serialize()`* method or use *`simfile.mutate()`* for a more guided workflow.

3.7.4 Finding charts

In 1.0, the list of charts at `Simfile.charts` offered convenience methods for getting a single chart or finding multiple charts:

```
>>> from simfile import Simfile # 1.0
>>> sm = Simfile('testdata/nekonabe/nekonabe.sm')
>>> single_novice = sm.charts.get(difficulty='Beginner')
>>> single_novice.stepstype
dance-single
>>> expert_charts = sm.charts.filter(difficulty='Challenge')
>>> [ex.stepstype for ex in expert_charts]
['dance-double', 'dance-single']
```

In 2.0, these convenience methods have been removed in favor of for-loops and the built-in `filter` function. Writing your own predicates as Python code is much more flexibile than the 1.0 convenience methods, which could only find charts by exact property matches.

3.7.5 Special property types

In 1.0, certain properties of simfiles and charts were automatically converted from strings to richer representations.

- The "BPMS" and "STOPS" simfile parameters were converted to `Timing` objects that offered convenient access to the beat & value pairs:

  ```
  >>> from simfile import Simfile # 1.0
  >>> sm = Simfile('testdata/nekonabe/nekonabe.sm')
  >>> print(type(sm['BPMS']))
  <class 'simfile.simfile.Timing'>
  >>> print(type(sm['STOPS']))
  <class 'simfile.simfile.Timing'>
  ```

- The "meter" and "notes" chart attributes were converted to an integer and a `Notes` object, respectively:

  ```
  >>> from simfile import Simfile # 1.0
  >>> sm = Simfile('testdata/nekonabe/nekonabe.sm')
  >>> chart = sm.charts[0]
  >>> print(type(chart.meter))
  ```

(continues on next page)

(continued from previous page)

```
<class 'int'>
>>> print(type(chart.notes))
<class 'simfile.simfile.Notes'>
```

In 2.0, all properties of simfiles and charts are kept as strings. This prevents wasting CPU cycles for use cases that don't benefit from the richer representations, keeps the underlying data structures homogeneously typed, and significantly reduces the number of reasons why parsing a simfile might fail.

If you need rich timing data, use the *simfile.timing* package:

```
>>> import simfile # 2.0
>>> from simfile.timing import TimingData
>>> nekonabe = simfile.open('testdata/nekonabe/nekonabe.sm')
>>> timing_data = TimingData(nekonabe)
>>> print(timing_data.bpms[0])
BeatValue(beat=Beat(0), value=Decimal('150.000'))
```

If you need rich note data, use the *simfile.notes* package:

```
>>> import simfile # 2.0
>>> from simfile.notes import NoteData
>>> from simfile.timing import Beat
>>> nekonabe = simfile.open('testdata/nekonabe/nekonabe.sm')
>>> for note in NoteData(nekonabe.charts[0]):
...     if note.beat > Beat(18): break
...     print(note)
...
Note(beat=Beat(16.25), column=3, note_type=NoteType.TAP)
Note(beat=Beat(16.5), column=2, note_type=NoteType.TAP)
Note(beat=Beat(17.25), column=2, note_type=NoteType.TAP)
Note(beat=Beat(17.5), column=3, note_type=NoteType.TAP)
```

Keeping these modules separate from the core simfile & chart classes enables them to be much more fully-featured than their 1.0 counterparts.

3.8 API Reference

This page contains auto-generated API reference documentation[1].

3.8.1 simfile

Convenience functions for loading & modifying simfiles.

All functions take a *strict* parameter that defaults to True. By default, the underlying parser will throw an exception if it finds any stray text between parameters. This behavior can be overridden by setting *strict* to False.

[1] Created with sphinx-autoapi

Subpackages

simfile.notes

Note data classes, plus submodules that operate on note data.

Submodules

simfile.notes.count

Module Contents

Functions

count_grouped_notes(grouped_notes_iterator: Iterable[simfile.notes.group.GroupedNotes], same_beat_minimum: int = 1) → int	Count a stream of `GroupedNotes`.
count_steps(notes: Iterable[simfile.notes.Note], *, include_note_types: FrozenSet[simfile.notes.NoteType] = DEFAULT_NOTE_TYPES, same_beat_notes: simfile.notes.group.SameBeatNotes = SameBeatNotes.JOIN_ALL, same_beat_minimum: int = 1) → int	Count the steps in a note stream.
count_jumps(notes: Iterable[simfile.notes.Note], *, include_note_types: FrozenSet[simfile.notes.NoteType] = DEFAULT_NOTE_TYPES, same_beat_notes: simfile.notes.group.SameBeatNotes = SameBeatNotes.JOIN_ALL) → int	Count the jumps (2+ simultaneous notes) in a note stream.
count_mines(notes: Iterable[simfile.notes.Note]) → int	Count the mines in a note stream.
count_hands(notes: Iterable[simfile.notes.Note], *, include_note_types: FrozenSet[simfile.notes.NoteType] = DEFAULT_NOTE_TYPES, same_beat_notes: simfile.notes.group.SameBeatNotes = SameBeatNotes.JOIN_ALL, same_beat_minimum: int = 3) → int	Count the hands (3+ simultaneous notes) in a note stream.
count_holds(notes: Iterable[simfile.notes.Note], *, orphaned_head: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION, orphaned_tail: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION) → int	Count the hold notes in a note stream.
count_rolls(notes: Iterable[simfile.notes.Note], *, orphaned_head: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION, orphaned_tail: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION) → int	Count the roll notes in a note stream.

simfile.notes.count.**count_grouped_notes**(*grouped_notes_iterator: Iterable[simfile.notes.group.GroupedNotes]*, *same_beat_minimum: int = 1*) → int

Count a stream of `GroupedNotes`.

To count only groups of N or more notes, use *same_beat_minimum*.

simfile.notes.count.**count_steps**(*notes: Iterable[simfile.notes.Note]*, *, include_note_types: FrozenSet[simfile.notes.NoteType] = DEFAULT_NOTE_TYPES*, *same_beat_notes: simfile.notes.group.SameBeatNotes = SameBeatNotes.JOIN_ALL*, *same_beat_minimum: int = 1*) → int

Count the steps in a note stream.

The definition of "step count" varies by application; the default configuration tries to match StepMania's definition as closely as possible:

- Taps, holds, rolls, and lifts are eligible for counting.
- Multiple inputs on the same beat are only counted once.

These defaults can be changed using the keyword parameters. Refer to `SameBeatNotes` for alternative ways to count same-beat notes.

simfile.notes.count.**count_jumps**(*notes: Iterable[simfile.notes.Note]*, *, include_note_types: FrozenSet[simfile.notes.NoteType] = DEFAULT_NOTE_TYPES*, *same_beat_notes: simfile.notes.group.SameBeatNotes = SameBeatNotes.JOIN_ALL*) → int

Count the jumps (2+ simultaneous notes) in a note stream.

This implementation defers to `count_steps()` with the same default parameters, except only groups of 2 or more notes are counted (i.e. *same_beat_minimum* is set to 2).

simfile.notes.count.**count_mines**(*notes: Iterable[simfile.notes.Note]*) → int

Count the mines in a note stream.

simfile.notes.count.**count_hands**(*notes: Iterable[simfile.notes.Note]*, *, include_note_types: FrozenSet[simfile.notes.NoteType] = DEFAULT_NOTE_TYPES*, *same_beat_notes: simfile.notes.group.SameBeatNotes = SameBeatNotes.JOIN_ALL*, *same_beat_minimum: int = 3*) → int

Count the hands (3+ simultaneous notes) in a note stream.

This implementation defers to `count_steps()` with the same default parameters, except only groups of 3 or more notes are counted (i.e. *same_beat_minimum* is set to 3).

simfile.notes.count.**count_holds**(*notes: Iterable[simfile.notes.Note]*, *, orphaned_head: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*, *orphaned_tail: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*) → int

Count the hold notes in a note stream.

By default, this method validates that hold heads connect to their corresponding tails. This validation can be turned off by setting the *orphaned_head* and *orphaned_tail* arguments to *KEEP_ORPHAN* or *DROP_ORPHAN*; see `OrphanedNotes` for more details.

simfile.notes.count.**count_rolls**(*notes: Iterable[simfile.notes.Note]*, *, *orphaned_head: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*, *orphaned_tail: simfile.notes.group.OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*) → int

Count the roll notes in a note stream.

By default, this method validates that roll heads connect to their corresponding tails. This validation can be turned off by setting the *orphaned_head* and *orphaned_tail* arguments to *KEEP_ORPHAN* or *DROP_ORPHAN*; see *OrphanedNotes* for more details.

simfile.notes.group

Module Contents

Classes

NoteWithTail	A hold/roll head note with its corresponding tail note.
SameBeatNotes	Choices for *group_notes()*' *same_beat_notes* parameter.
OrphanedNotes	Choices for *group_notes()*' *orphaned_head|tail* parameters.

Functions

group_notes(notes: Iterable[simfile.notes.Note], *, include_note_types: FrozenSet[simfile.notes.NoteType] = frozenset(NoteType), same_beat_notes: SameBeatNotes = SameBeatNotes.KEEP_SEPARATE, join_heads_to_tails: bool = False, orphaned_head: OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION, orphaned_tail: OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION) → Iterator[GroupedNotes]	Group notes that are often considered linked to one another.
ungroup_notes(grouped_notes: Iterable[GroupedNotes], *, orphaned_notes: OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION) → Iterator[simfile.notes.Note]	Convert grouped notes back into a plain note stream.

Attributes

GroupedNotes	A sequence of *Note* and possibly *NoteWithTail* objects.

class simfile.notes.group.**NoteWithTail**
Bases: NamedTuple

A hold/roll head note with its corresponding tail note.

beat :simfile.timing.Beat

column :int

note_type :simfile.notes.NoteType

tail_beat :simfile.timing.Beat

player :int = 0
Only used in routine charts. The second player's note data will have this value set to 1.

keysound_index :Optional[int]
Only used in keysounded SSC charts. Notes followed by a number in square brackets will have this value set to the bracketed number.

simfile.notes.group.**GroupedNotes**
A sequence of *Note* and possibly *NoteWithTail* objects.

class simfile.notes.group.**SameBeatNotes**
Bases: enum.Enum

Choices for *group_notes()*' *same_beat_notes* parameter.

When multiple notes land on the same beat...

- *KEEP_SEPARATE*: each note is emitted separately
- *JOIN_BY_NOTE_TYPE*: notes of the same type are emitted together
- *JOIN_ALL*: all notes are emitted together

KEEP_SEPARATE = 1

JOIN_BY_NOTE_TYPE = 2

JOIN_ALL = 3

exception simfile.notes.group.**OrphanedNoteException**
Bases: Exception

Raised by *group_notes()* to flag an orphaned head or tail note.

class simfile.notes.group.**OrphanedNotes**
Bases: enum.Enum

Choices for *group_notes()*' *orphaned_head\tail* parameters.

When *join_heads_to_tails* is True and a head or tail note is missing its counterpart...

- *RAISE_EXCEPTION*: raise *OrphanedNoteException*
- *KEEP_ORPHAN*: emit the orphaned Note
- *DROP_ORPHAN*: do not emit the orphaned note

RAISE_EXCEPTION = 1

KEEP_ORPHAN = 2

DROP_ORPHAN = 3

simfile.notes.group.**group_notes**(*notes: Iterable[simfile.notes.Note]*, ***, *include_note_types: FrozenSet[simfile.notes.NoteType] = frozenset(NoteType)*, *same_beat_notes: SameBeatNotes = SameBeatNotes.KEEP_SEPARATE*, *join_heads_to_tails: bool = False*, *orphaned_head: OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*, *orphaned_tail: OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*) → Iterator[*GroupedNotes*]

Group notes that are often considered linked to one another.

There are two kinds of connected notes: notes that occur on the same beat ("jumps") and hold/roll notes with their corresponding tails. Either or both of these connection types can be opted into using the constructor parameters.

Generators produced by this class yield *GroupedNotes* objects, rather than Note objects. These are sequences that generally contain Note and *NoteWithTail* objects, although the output may be more restrained depending on the configuration.

When *join_heads_to_tails* is set to True, tail notes are attached to their corresponding hold/roll heads as *NoteWithTail* objects. The tail itself will not be emitted as a separate note. If a head or tail note is missing its counterpart, *orphaned_head* and *orphaned_tail* determine the behavior. (These parameters are ignored if *join_heads_to_tails* is omitted or False.)

Refer to each enum's documentation for the other configuration options.

simfile.notes.group.**ungroup_notes**(*grouped_notes: Iterable[GroupedNotes]*, ***, *orphaned_notes: OrphanedNotes = OrphanedNotes.RAISE_EXCEPTION*) → Iterator[*simfile.notes.Note*]

Convert grouped notes back into a plain note stream.

If a note falls within a *NoteWithTail*'s head and tail (on the same column), it would cause the head and tail to be orphaned. *orphaned_notes* determines how to handle the splitting note: *KEEP_ORPHAN* will yield the note (allowing the head and tail notes to become orphans) and *DROP_ORPHAN* will drop the note (preserving the link between the head and tail notes).

Note that this check only applies to heads and tails joined as a NoteAndTail. If *group_notes()* was called without specifying *join_heads_to_tails*, specifying *orphaned_notes* here will have no effect. This mirrors how *group_notes()*' *orphaned_head* and *orphaned_tail* parameters behave.

simfile.notes.timed

Module Contents

Classes

TimedNote	A note with its song time attached.
UnhittableNotes	How to handle timed notes that are unhittable due to warp segments.

Functions

time_notes(note_data: simfile.notes.NoteData, timing_data: simfile.timing.TimingData, unhittable_notes: UnhittableNotes = UnhittableNotes.TAP_TO_FAKE) → Iterator[TimedNote]	Generate a stream of timed notes from the supplied note & timing data.

class simfile.notes.timed.**TimedNote**
Bases: NamedTuple

A note with its song time attached.

time :simfile.timing.engine.SongTime

note :simfile.notes.Note

class simfile.notes.timed.**UnhittableNotes**
Bases: enum.Enum

How to handle timed notes that are unhittable due to warp segments.

When a note is unhittable...

- *TAP_TO_FAKE*: convert tap notes to fakes, drop other note types
- *DROP_NOTE*: drop the unhittable note regardless of type
- *KEEP_NOTE*: keep the unhittable note

TAP_TO_FAKE = 1

DROP_NOTE = 2

KEEP_NOTE = 3

simfile.notes.timed.**time_notes**(*note_data:* simfile.notes.NoteData, *timing_data:* simfile.timing.TimingData, *unhittable_notes:* UnhittableNotes = *UnhittableNotes.TAP_TO_FAKE*) → Iterator[*TimedNote*]
Generate a stream of timed notes from the supplied note & timing data.

For notes that are unhittable due to warps, the *unhittable_notes* parameter determines the behavior. See *UnhittableNotes* for more details.

Package Contents

Classes

NoteType	Known note types supported by StepMania.
Note	A note, corresponding to a nonzero character in a chart's note data.
NoteData	Wrapper for note data with iteration & serialization capabilities.

class simfile.notes.**NoteType**
Bases: enum.Enum

Known note types supported by StepMania.

TAP = 1

HOLD_HEAD = 2

TAIL = 3

ROLL_HEAD = 4

ATTACK = A

FAKE = F

KEYSOUND = K

LIFT = L

MINE = M

class simfile.notes.**Note**
Bases: NamedTuple

A note, corresponding to a nonzero character in a chart's note data.

Note objects are intrinsically ordered according to their position in the underlying note data: that is, if *note1* would appear earlier in the note data string than *note2*, then *note1* < *note2* is true.

beat :simfile.timing.Beat

column :int

note_type :NoteType

player :int = 0
Only used in routine charts. The second player's note data will have this value set to 1.

keysound_index :Optional[int]
Only used in keysounded SSC charts. Notes followed by a number in square brackets will have this value set to the bracketed number.

class simfile.notes.**NoteData**(*source: Union[str,* simfile.types.Chart*,* NoteData*]*)
Wrapper for note data with iteration & serialization capabilities.

The constructor accepts a string of note data, any *Chart*, or another *NoteData* instance.

property columns(*self*)
How many note columns this chart has.

classmethod from_notes(*cls: Type[*NoteData*]*, *notes: Iterable[*Note*]*, *columns: int*) → *NoteData*
Convert a stream of notes into note data.

This method assumes the following preconditions:

- The input notes are naturally sorted.
- Every note's beat is nonnegative.
- Every note's column is nonnegative and less than *columns*.

Note that this method doesn't quantize beats to 192nd ticks, and off-grid notes may result in measures with more rows than the StepMania editor would produce. StepMania will quantize these notes gracefully during gameplay, but you can apply *Beat.round_to_tick()* to each note's beat if you'd prefer to keep the note data tidy.

simfile.timing

Timing data classes, plus submodules that operate on timing data.

Submodules

simfile.timing.displaybpm

Function & cla

Module Contents

Classes

StaticDisplayBPM	A single BPM value.
RangeDisplayBPM	A range of BPM values.
RandomDisplayBPM	Used by StepMania to obfuscate the displayed BPM with random numbers.

Functions

displaybpm(simfile: simfile.types.Simfile, ssc_chart: simfile.ssc.SSCChart = SSCChart(), ignore_specified: Optional[bool] = False) → DisplayBPM	Get the display BPM from a simfile and optionally an SSC chart.

Attributes

DisplayBPM	Type union of *StaticDisplayBPM*, *RangeDisplayBPM*, and *RandomDisplayBPM*.

class simfile.timing.displaybpm.**StaticDisplayBPM**
Bases: NamedTuple

A single BPM value.

value :decimal.Decimal
The single BPM value. This property is None in the other DisplayBPM classes.

property min(*self*) → decimal.Decimal
Returns the single BPM value.

property max(*self*) → decimal.Decimal
Returns the single BPM value.

property range(*self*) → None

class simfile.timing.displaybpm.**RangeDisplayBPM**
Bases: NamedTuple

A range of BPM values.

min :decimal.Decimal

max :decimal.Decimal

property value(*self*) → None

property range(*self*) → Tuple[decimal.Decimal, decimal.Decimal]
(min, max) tuple. This property is None in the other DisplayBPM classes.

class simfile.timing.displaybpm.**RandomDisplayBPM**
Bases: NamedTuple

Used by StepMania to obfuscate the displayed BPM with random numbers.

property value(*self*) → None

property min(*self*) → None

property max(*self*) → None

property range(*self*) → None

simfile.timing.displaybpm.**DisplayBPM**
Type union of *StaticDisplayBPM*, *RangeDisplayBPM*, and *RandomDisplayBPM*.

simfile.timing.displaybpm.**displaybpm**(*simfile:* simfile.types.Simfile, *ssc_chart:* simfile.ssc.SSCChart = *SSCChart()*, *ignore_specified:* *Optional[bool] = False*) → *DisplayBPM*
Get the display BPM from a simfile and optionally an SSC chart.

If a DISPLAYBPM property is present (and *ignore_specified* isn't set to True), its value is used as follows:

- One number maps to *StaticDisplayBPM*
- Two ":"-separated numbers maps to *RangeDisplayBPM*
- A literal "*" maps to *RandomDisplayBPM*

Otherwise, the BPMS property will be used. A single BPM maps to *StaticDisplayBPM*; if there are multiple, the minimum and maximum will be identified and passed to *RangeDisplayBPM*.

If both an *SSCSimfile* (version 0.7 or higher) and an *SSCChart* are provided, and if the chart contains any timing fields, the chart will be used as the source of timing.

simfile.timing.engine

Module Contents

Classes

SongTime	A floating-point time value, denoting a temporal position in a simfile.
EventTag	Types of timing events.
TimingEngine	Convert song time to beats and vice-versa.

class simfile.timing.engine.**SongTime**
Bases: float

A floating-point time value, denoting a temporal position in a simfile.

class simfile.timing.engine.**EventTag**
Bases: enum.IntEnum

Types of timing events.

The order of these values determines how multiple events on the same beat will be sorted: for example, delays must occur before stops in order to correctly time notes on a beat with both a delay and a stop.

Warps, delays, and stops have a corresponding "end" type that *TimingEngine* uses to simplify the beat/time conversion logic. These can be used to disambiguate the time at a given beat (for stops & delays) or the beat at a given time (for warps).

WARP = 0

WARP_END = 1

BPM = 2

DELAY = 3

DELAY_END = 4

STOP = 5

STOP_END = 6

class simfile.timing.engine.**TimingEngine**(*timing_data:* simfile.timing.TimingData)
Convert song time to beats and vice-versa.

Under the hood, this class arranges timing events chronologically, determines the song time and BPM at each event, then extrapolates from those calculated values for each *bpm_at()* / *time_at()* / *beat_at()* call.

timing_data :simfile.timing.TimingData

bpm_at(*self*, *beat:* simfile.timing.Beat) → decimal.Decimal
Find the song's BPM at a given beat.

Neither warps, stops, nor delays affect the output of this method: warps are not considered "infinite BPM", nor are pauses considered "zero BPM".

hittable(*self*, *beat:* simfile.timing.Beat) → bool
Determine if a note on the given beat would be hittable.

A note is considered "unhittable" if and only if:

- It takes place inside a warp segment (inclusive of the warp's start, exclusive of the warp's end).
- It doesn't coincide with a stop or delay.

StepMania internally converts unhittable notes to fake notes so that the player's score isn't affected by them.

time_at(*self*, *beat:* simfile.timing.Beat, *event_tag:* EventTag = *EventTag.STOP*) → *SongTime*
Determine the song time at a given beat.

On most beats, the *event_tag* parameter is inconsequential. The only time it matters is when stops or delays are involved:

- On stops, providing a value of *EventTag.STOP* or lower will return the time at which the stop is reached, whereas providing *EventTag.STOP_END* will return the time when the stop ends.
- On delays, providing a value of *EventTag.DELAY* or lower will return the time at which the delay is reached, whereas providing *EventTag.DELAY_END* or later will return the time when the delay ends.

The default value of *EventTag.STOP* effectively matches the time at which a note on the given beat must be hit (assuming such a note is *hittable()*).

beat_at (*self*, *time: SongTimeOrFloat*, *event_tag:* EventTag = *EventTag.STOP*) → *simfile.timing.Beat*
Determine the beat at a given time in the song.

At most times, the *event_tag* parameter is inconsequential. The only time it matters is when the time lands exactly on a warp segment:

- Providing *EventTag.WARP* will return the beat where the warp starts.
- Providing *EventTag.WARP_END* or later will return the beat where the warp ends (or is interrupted by a stop or delay).

Keep in mind that this situation is floating-point precise, so it's unlikely for the *event_tag* to ever make a difference.

Package Contents

Classes

Beat	A fractional beat value, denoting vertical position in a simfile.
BeatValue	An event that occurs on a particular beat, e.g. a BPM change or stop.
BeatValues	A list of *BeatValue* instances.
TimingData	Timing data for a simfile, possibly enriched with SSC chart timing.

class simfile.timing.**Beat**
Bases: fractions.Fraction

A fractional beat value, denoting vertical position in a simfile.

The constructor the same arguments as Python's Fraction:

Takes a string like '3/2' or '1.5', another Rational instance, a numerator/denominator pair, or a float.

If the input is a float or string, the resulting fraction will be rounded to the nearest *tick()*.

classmethod tick (*cls*) → *Beat*
1/48 of a beat (1/192 of a measure).

classmethod from_str (*cls*, *beat_str*) → *Beat*
Convert a decimal string to a beat, rounding to the nearest tick.

round_to_tick (*self*) → *Beat*
Round the beat to the nearest tick.

class simfile.timing.**BeatValue**
Bases: NamedTuple

An event that occurs on a particular beat, e.g. a BPM change or stop.

The decimal value's semantics vary based on the type of event:

- BPMS: the new BPM value
- STOPS, DELAYS: number of seconds to pause

- WARPS: number of beats to skip

beat :Beat

value :decimal.Decimal

class simfile.timing.**BeatValues**(*initlist=None*)
Bases: simfile._private.generic.ListWithRepr[*BeatValue*]

A list of *BeatValue* instances.

classmethod from_str(*cls: Type[*BeatValues*]*, *string: Optional[str]*) → *BeatValues*
Parse the MSD value component of a timing data list.

Specifically, *BPMS*, *STOPS*, *DELAYS*, and *WARPS* are the timing data lists whose values can be parsed by this method.

class simfile.timing.**TimingData**(*simfile: simfile.types.Simfile*, *chart: Optional[*simfile.types.Chart*] = None*)
Timing data for a simfile, possibly enriched with SSC chart timing.

If both an *SSCSimfile* (version 0.7 or higher) and an *SSCChart* are supplied to the constructor, and if the chart contains any timing fields, the chart will be used as the source of timing data.

Per StepMania's behavior, the offset defaults to zero if the simfile (and/or SSC chart) doesn't specify one. (However, unlike StepMania, the BPM does not default to 60 when omitted; the default BPM doesn't appear to be used deliberately in any existing simfiles, whereas the default offset does get used intentionally from time to time.)

bpms :BeatValues

stops :BeatValues

delays :BeatValues

warps :BeatValues

offset :decimal.Decimal

Submodules

simfile.assets

Module Contents

Classes

Assets	Asset loader for a simfile directory.

class simfile.assets.**Assets**(*simfile_dir: str*, ***, *simfile: Optional[*simfile.types.Simfile*] = None*, *filesystem: fs.base.FS = NativeOSFS()*, ***kwargs*)
Asset loader for a simfile directory.

This loader uses the same heuristics as StepMania to determine a default path for assets not specified in the simfile. For example, if the BACKGROUND property is missing or blank, StepMania will look for an image whose filename contains "background" or ends with "bg".

The simfile will be loaded from the simfile_dir unless an explicit simfile argument is supplied. This is intended mostly as an optimization for when the simfile has already been loaded from disk.

All asset paths are absolute and normalized. Keyword arguments are passed down to *simfile.open()* (and are only valid if `simfile` is not provided).

property music(*self*)

property banner(*self*)

property background(*self*)

property cdtitle(*self*)

property jacket(*self*)

property cdimage(*self*)

property disc(*self*)

simfile.base

Base classes for simfile & chart implementations.

This module should ideally never need to be used directly, but its documentation may be useful for understanding the similarities between the SM and SSC formats.

Module Contents

Classes

BaseChart	One chart from a simfile.
BaseCharts	List containing all of a simfile's charts.
BaseSimfile	A simfile, including its metadata (e.g. song title) and charts.

class simfile.base.**BaseChart**

Bases: `collections.OrderedDict`, `simfile._private.serializable.Serializable`

One chart from a simfile.

All charts have the following known properties: *stepstype*, *description*, *difficulty*, *meter*, *radarvalues*, and *notes*.

stepstype

description

difficulty

meter

radarvalues

notes

classmethod blank(*cls*)

Generate a blank, valid chart populated with standard keys.

This should approximately match blank charts produced by the StepMania editor.

class simfile.base.**BaseCharts**(*data=None*)

Bases: `simfile._private.generic.ListWithRepr[simfile._private.generic.E]`,

simfile._private.serializable.Serializable

List containing all of a simfile's charts.

serialize(*self*, *file: TextIO*)
Write the object to provided text file object as MSD.

class simfile.base.**BaseSimfile**(*, *file: Optional[Union[TextIO, Iterator[str]]] = None*, *string: Optional[str] = None*, *strict: bool = True*)
Bases: collections.OrderedDict, simfile._private.serializable.Serializable

A simfile, including its metadata (e.g. song title) and charts.

Metadata is stored directly on the simfile object through a dict-like interface. Keys are unique (if there are duplicates, the last value wins) and converted to uppercase.

Additionally, properties recognized by the current stable version of StepMania are exposed through lower-case properties on the object for easy (and implicitly spell-checked) access. The following known properties are defined:

- Metadata: *title*, *subtitle*, *artist*, *titletranslit*, *subtitletranslit*, *artisttranslit*, *genre*, *credit*, *samplestart*, *samplelength*, *selectable*, *instrumenttrack*, *timesignatures*
- File paths: *banner*, *background*, *lyricspath*, *cdtitle*, *music*
- Gameplay events: *bgchanges*, *fgchanges*, *keysounds*, *attacks*, *tickcounts*
- Timing data: *offset*, *bpms*, *stops*, *delays*

If a desired simfile property isn't in this list, it can still be accessed as a dict item.

By default, the underlying parser will throw an exception if it finds any stray text between parameters. This behavior can be overridden by setting *strict* to False in the constructor.

MULTI_VALUE_PROPERTIES = ['ATTACKS', 'DISPLAYBPM']

title

subtitle

artist

titletranslit

subtitletranslit

artisttranslit

genre

credit

banner

background

lyricspath

cdtitle

music

offset

bpms

stops

delays

timesignatures

tickcounts

instrumenttrack

samplestart

samplelength

displaybpm

selectable

bgchanges

fgchanges

keysounds

attacks

property charts(*self*) → *BaseCharts*
List of charts associated with this simfile.

classmethod blank(*cls*)
Generate a blank, valid simfile populated with standard keys.

This should approximately match the simfile produced by the StepMania editor in a directory with no .sm or .ssc files.

serialize(*self*, *file: TextIO*)
Write the object to provided text file object as MSD.

simfile.convert

Functions for converting SM to SSC simfiles and vice-versa.

Module Contents

Classes

PropertyType	Types of known properties.
InvalidPropertyBehavior	How to handle an invalid property during conversion.

Functions

sm_to_ssc(sm_simfile: simfile.sm.SMSimfile, *, simfile_template: Optional[simfile.ssc.SSCSimfile] = None, chart_template: Optional[simfile.ssc.SSCChart] = None) → simfile.ssc.SSCSimfile	Convert an SM simfile to an equivalent SSC simfile.

continues on next page

Table 17 – continued from previous page

ssc_to_sm(ssc_simfile: simfile.ssc.SSCSimfile, *, simfile_template: Optional[simfile.sm.SMSimfile] = None, chart_template: Optional[simfile.sm.SMChart] = None, invalid_property_behaviors: InvalidPropertyBehaviorMapping = {}) → simfile.sm.SMSimfile	Convert an SSC simfile to an equivalent SM simfile.

class simfile.convert.**PropertyType**

Bases: enum.Enum

Types of known properties.

These roughly mirror the lists of known properties documented in *BaseSimfile* and *SSCSimfile*.

SSC_VERSION = 1

The SSC version tag.

METADATA = 2

Properties that don't affect the gameplay.

FILE_PATH = 3

Properties that reference file paths (e.g. images).

GAMEPLAY_EVENT = 4

Properties that affect gameplay in some fashion.

TIMING_DATA = 5

Properties that influence when notes must be hit.

class simfile.convert.**InvalidPropertyBehavior**

Bases: enum.Enum

How to handle an invalid property during conversion.

COPY_ANYWAY = 1

Copy the property regardless of the destination type.

IGNORE = 2

Do not copy the property.

ERROR_UNLESS_DEFAULT = 3

Raise *InvalidPropertyException* unless the property's value is the default for its field.

The "default value" for most properties is an empty string. If the destination type's `.blank` output has a non-empty value for the property, that value is considered the default instead.

ERROR = 4

Raise *InvalidPropertyException* regardless of the value.

exception simfile.convert.**InvalidPropertyException**

Bases: Exception

Raised by conversion functions if a property cannot be converted.

simfile.convert.**sm_to_ssc**(*sm_simfile:* simfile.sm.SMSimfile, *, *simfile_template:* *Optional[*simfile.ssc.SSCSimfile*]* = *None*, *chart_template:* *Optional[*simfile.ssc.SSCChart*]* = *None*) → *simfile.ssc.SSCSimfile*

Convert an SM simfile to an equivalent SSC simfile.

simfile_template and *chart_template* can optionally be provided to define the initial simfile and chart prior to copying properties from the source object. If they are not provided, *SSCSimfile.blank()* and *SSCChart.blank()* will supply the template objects.

simfile.convert.**ssc_to_sm**(*ssc_simfile:* simfile.ssc.SSCSimfile, *, *simfile_template:* Optional[simfile.sm.SMSimfile] = *None*, *chart_template:* Optional[simfile.sm.SMChart] = *None*, *invalid_property_behaviors:* InvalidPropertyBehaviorMapping = *{}*) → *simfile.sm.SMSimfile*

Convert an SSC simfile to an equivalent SM simfile.

simfile_template and *chart_template* can optionally be provided to define the initial simfile and chart prior to copying properties from the source object. If they are not provided, `SMSimfile.blank()` and `SMChart.blank()` will supply the template objects.

Not all SSC properties are valid for SM simfiles, including some gameplay events and timing data. If one of those types of properties are found and contain a non-default value, `InvalidPropertyException` will be raised.

The behavior described above can be changed by supplying the *invalid_property_behaviors* parameter, which maps `PropertyType` to `InvalidPropertyBehavior` values. This mapping need not cover every `PropertyType`; any missing values will fall back to the default mapping described above.

simfile.dir

Module Contents

Classes

`SimfileDirectory`	A simfile directory, containing an SM and/or SSC file, or neither.
`SimfilePack`	A simfile pack directory, containing any number of simfile directories.

exception simfile.dir.**DuplicateSimfileError**

Bases: Exception

Raised when a simfile directory contains multiple simfiles of the same type (e.g. two SM files).

class simfile.dir.**SimfileDirectory**(*simfile_dir: str*, *, *filesystem: fs.base.FS = NativeOSFS()*, *ignore_duplicate=False*)

A simfile directory, containing an SM and/or SSC file, or neither.

Raises `DuplicateSimfileError` if the directory contains multiple simfiles of the same type (e.g. two SM files).

sm_path :Optional[str]

Absolute path to the SM file, if present.

ssc_path :Optional[str]

Absolute path to the SSC file, if present.

property simfile_path(*self*)

The SSC path if present, otherwise the SM path.

open(*self*, ***kwargs*) → *simfile.types.Simfile*

Open the simfile in this directory.

If both SSC and SM are present, SSC is preferred. Keyword arguments are passed down to `simfile.open()`.

Raises `FileNotFoundError` if there is no SM or SSC file in the directory.

assets (*self*) → *simfile.assets.Assets*
Get the file assets for this simfile.

class simfile.dir.**SimfilePack**(*pack_dir: str*, *, *filesystem: fs.base.FS = NativeOSFS()*, *ignore_duplicate: bool = False*)
A simfile pack directory, containing any number of simfile directories.

Only immediate subdirectories of `pack_dir` containing an SM or SSC file are included. Simfiles aren't guaranteed to appear in any particular order.

simfile_dir_paths :Tuple[str]
Absolute paths to the simfile directories in this pack.

simfile_dirs (*self*) → Iterator[*SimfileDirectory*]
Iterator over the simfile directories in the pack.

simfiles (*self*, ***kwargs*) → Iterator[*simfile.types.Simfile*]
Iterator over the simfiles in the pack.

If both SSC and SM are present in a simfile directory, SSC is preferred. Keyword arguments are passed down to *simfile.open()*.

property name (*self*)
Get the name of the pack (the directory name by itself).

banner (*self*) → Optional[str]
Get the pack's banner image, if present, as an absolute path.

Follows the same logic as StepMania:

- When there are multiple images in the pack directory, the banner is chosen first by extension priority (PNG is highest, then JPG, JPEG, GIF, BMP), then alphabetically.
- If there are no images in the pack directory, checks for a banner *alongside* the pack with the same base name, using the same extension priority as before.

simfile.sm

Simfile & chart classes for SM files.

Module Contents

Classes

SMChart	SM implementation of *BaseChart*.
SMCharts	SM implementation of *BaseCharts*.
SMSimfile	SM implementation of *BaseSimfile*.

class simfile.sm.**SMChart**
Bases: *simfile.base.BaseChart*

SM implementation of *BaseChart*.

Unlike *SSCChart*, SM chart metadata is stored as a fixed list of 6 properties, so this class prohibits adding or deleting keys from its backing OrderedDict.

extradata :Optional[List[str]]
If the chart data contains more than 6 components, the extra components will be stored in this attribute.

classmethod blank(*cls: Type[*SMChart*]*) → *SMChart*
Generate a blank, valid chart populated with standard keys.

This should approximately match blank charts produced by the StepMania editor.

classmethod from_str(*cls: Type[*SMChart*]*, *string: str*) → *SMChart*
Parse the serialized MSD value components of a NOTES property.

The string should contain six colon-separated components, corresponding to each of the base known properties documented in *BaseChart*. Any additional components will be stored in *extradata*.

Raises ValueError if the string contains fewer than six components.

Deprecated since version 2.1: This is now a less efficient version of *from_msd()*, which interoperates better with msdparser version 2.0.

classmethod from_msd(*cls: Type[*SMChart*]*, *values: Sequence[str]*) → *SMChart*
Parse the MSD value components of a NOTES property.

The list should contain six strings, corresponding to each of the base known properties documented in *BaseChart*. Any additional components will be stored in *extradata*.

Raises ValueError if the list contains fewer than six components.

serialize(*self*, *file*)
Write the object to provided text file object as MSD.

abstract update(*self*, **args*, ***kwargs*) → None
Raises NotImplementedError.

abstract pop(*self*, *property*, *default=None*)
Raises NotImplementedError.

abstract popitem(*self*, *last=True*)
Raises NotImplementedError.

class simfile.sm.**SMCharts**(*data=None*)
Bases: *simfile.base.BaseCharts*[*SMChart*]

SM implementation of *BaseCharts*.

List elements are *SMChart* instances.

class simfile.sm.**SMSimfile**(**, file: Optional[Union[TextIO, Iterator[str]]] = None, string: Optional[str] = None, strict: bool = True*)
Bases: *simfile.base.BaseSimfile*

SM implementation of *BaseSimfile*.

stops
Specialized property for *STOPS* that supports *FREEZES* as an alias.

classmethod blank(*cls: Type[*SMSimfile*]*) → *SMSimfile*
Generate a blank, valid simfile populated with standard keys.

This should approximately match the simfile produced by the StepMania editor in a directory with no .sm or .ssc files.

property charts(*self*) → *SMCharts*
List of charts associated with this simfile.

simfile.ssc

Simfile & chart classes for SSC files.

Module Contents

Classes

SSCChart	SSC implementation of *BaseChart*.
SSCCharts	SSC implementation of *BaseCharts*.
SSCSimfile	SSC implementation of *BaseSimfile*.

class simfile.ssc.**SSCChart**

Bases: *simfile.base.BaseChart*

SSC implementation of *BaseChart*.

Unlike *SMChart*, SSC chart metadata is stored as key-value pairs, so this class allows full modification of its backing OrderedDict.

Adds the following known properties:

- Metadata: *chartname*, *chartstyle*, *credit*, *timesignatures*
- File paths: *music*
- Gameplay events: *tickcounts*, *combos*, *speeds*, *scrolls*, *fakes*, *attacks*
- Timing data: *bpms*, *stops*, *delays*, *warps*, *labels*, *offset*, *displaybpm*

chartname

chartstyle

credit

music

bpms

stops

delays

timesignatures

tickcounts

combos

warps

speeds

scrolls

fakes

labels

attacks

offset

displaybpm

notes

classmethod from_str(*cls: Type[*SSCChart*]*, *string: str*, *strict: bool = True*) → *SSCChart*
Parse a string containing MSD data into an SSC chart.

The first property's key must be *NOTEDATA*. Parsing ends at the *NOTES* (or *NOTES2*) property.

By default, the underlying parser will throw an exception if it finds any stray text between parameters. This behavior can be overridden by setting *strict* to False.

classmethod blank(*cls: Type[*SSCChart*]*) → *SSCChart*
Generate a blank, valid chart populated with standard keys.

This should approximately match blank charts produced by the StepMania editor.

serialize(*self*, *file*)
Write the object to provided text file object as MSD.

class simfile.ssc.**SSCCharts**(*data=None*)
Bases: *simfile.base.BaseCharts*[*SSCChart*]

SSC implementation of *BaseCharts*.

List elements are *SSCChart* instances.

class simfile.ssc.**SSCSimfile**(*, *file: Optional[Union[TextIO, Iterator[str]]] = None*, *string: Optional[str] = None*, *strict: bool = True*)
Bases: *simfile.base.BaseSimfile*

SSC implementation of *BaseSimfile*.

Adds the following known properties:

- SSC version: *version*
- Metadata: *origin*, *labels*, *musiclength*, *lastsecondhint*
- File paths: *previewvid*, *jacket*, *cdimage*, *discimage*, *preview*
- Gameplay events: *combos*, *speeds*, *scrolls*, *fakes*
- Timing data: *warps*

version

origin

previewvid

jacket

cdimage

discimage

preview

musiclength

lastsecondhint

warps

labels

combos

speeds

scrolls

fakes

classmethod blank(*cls: Type[*SSCSimfile*]*) → *SSCSimfile*
Generate a blank, valid simfile populated with standard keys.

This should approximately match the simfile produced by the StepMania editor in a directory with no .sm or .ssc files.

property charts(*self*) → *SSCCharts*
List of charts associated with this simfile.

simfile.types

Union types for simfile & chart classes.

Module Contents

simfile.types.**Simfile**
Union of *SSCSimfile* and *SMSimfile*.

simfile.types.**Charts**
Union of *SSCCharts* and *SMCharts*.

simfile.types.**Chart**
Union of *SSCChart* and *SMChart*.

Package Contents

Functions

load(file: Union[TextIO, Iterator[str]], strict: bool = True) → types.Simfile	Load a text file object as a simfile.
loads(string: str, strict: bool = True) → types.Simfile	Load a string containing simfile data as a simfile.
open(filename: str, strict: bool = True, filesystem: fs.base.FS = NativeOSFS(), **kwargs) → types.Simfile	Load a simfile by filename.
open_with_detected_encoding(filename: str, try_encodings: List[str] = ENCODINGS, strict: bool = True, filesystem: fs.base.FS = NativeOSFS(), **kwargs) → Tuple[types.Simfile, str]	Load a simfile by filename; returns the simfile and detected encoding.
opendir(simfile_dir: str, filesystem: fs.base.FS = NativeOSFS(), **kwargs) → Tuple[types.Simfile, str]	Open a simfile from its directory path;
openpack(pack_dir: str, filesystem: fs.base.FS = NativeOSFS(), **kwargs) → Iterator[Tuple[types.Simfile, str]]	Open a pack of simfiles from the pack's directory path;
mutate(input_filename: str, output_filename: Optional[str] = None, backup_filename: Optional[str] = None, try_encodings: List[str] = ENCODINGS, strict: bool = True, filesystem: fs.base.FS = NativeOSFS(), **kwargs) → Iterator[types.Simfile]	Context manager that loads & saves a simfile by filename.

simfile.**load**(*file: Union[TextIO, Iterator[str]]*, *strict: bool = True*) → *types.Simfile*

Load a text file object as a simfile.

If the file object has a filename with a matching extension, it will be used to infer the correct implementation. Otherwise, the first property in the file is peeked at. If its key is "VERSION", the file is treated as an SSC simfile; otherwise, it's treated as an SM simfile.

simfile.**loads**(*string: str*, *strict: bool = True*) → *types.Simfile*
Load a string containing simfile data as a simfile.

simfile.**open**(*filename: str*, *strict: bool = True*, *filesystem: fs.base.FS = NativeOSFS()*, ***kwargs*) → *types.Simfile*
Load a simfile by filename.

Keyword arguments are passed to the builtin *open* function. If no encoding is specified, this function will defer to *open_with_detected_encoding()*.

simfile.**open_with_detected_encoding**(*filename: str*, *try_encodings: List[str] = ENCODINGS*, *strict: bool = True*, *filesystem: fs.base.FS = NativeOSFS()*, ***kwargs*) → Tuple[*types.Simfile*, str]
Load a simfile by filename; returns the simfile and detected encoding.

Tries decoding the simfile as UTF-8, CP1252 (English), CP932 (Japanese), and CP949 (Korean), mirroring the encodings supported by StepMania. This list can be overridden by supplying *try_encodings*.

Keep in mind that no heuristics are performed to "guess" the correct encoding - this function simply tries each encoding in order until one succeeds. As such, non-UTF-8 files may successfully parse as the wrong encoding, resulting in *mojibake*. If you intend to write the simfile back to disk, make sure to use the same encoding that was detected to preserve the true byte sequence.

In general, you only need to use this function directly if you need to know the file's encoding. *open()* and *mutate()* both defer to this function to detect the encoding behind the scenes.

simfile.**opendir**(*simfile_dir: str*, *filesystem: fs.base.FS = NativeOSFS()*, ***kwargs*) → Tuple[*types.Simfile*, str]
Open a simfile from its directory path; returns a (simfile, filename) tuple.

If both SSC and SM are present, SSC is preferred. Keyword arguments are passed down to *simfile.open()*.

If you need more flexibility (for example, if you need both the SM and SSC files), try using *SimfileDirectory*.

simfile.**openpack**(*pack_dir: str*, *filesystem: fs.base.FS = NativeOSFS()*, ***kwargs*) → Iterator[Tuple[*types.Simfile*, str]]
Open a pack of simfiles from the pack's directory path; yields (simfile, filename) tuples.

Only immediate subdirectories of `pack_dir` containing an SM or SSC file are included. Simfiles aren't guaranteed to appear in any particular order. If both SSC and SM are present, SSC is preferred. Keyword arguments are passed down to *simfile.open()*.

If you need more flexibility (for example, if you need the pack's banner, or both the SM and SSC files), try using *SimfilePack*.

exception simfile.**CancelMutation**
Bases: `BaseException`

Raise from inside a *mutate()* block to prevent saving the simfile.

simfile.**mutate**(*input_filename: str*, *output_filename: Optional[str] = None*, *backup_filename: Optional[str] = None*, *try_encodings: List[str] = ENCODINGS*, *strict: bool = True*, *filesystem: fs.base.FS = NativeOSFS()*, ***kwargs*) → Iterator[*types.Simfile*]
Context manager that loads & saves a simfile by filename.

If an *output_filename* is provided, the modified simfile will be written to that filename upon exit from the context manager. Otherwise, it will be written back to the *input_filename*.

If a *backup_filename* is provided, the *original* simfile will be written to that filename upon exit from the context manager. Otherwise, no backup copy will be written. *backup_filename* must be distinct from *input_filename* and *output_filename* if present.

If the context manager catches an exception, nothing will be written to disk, and the exception will be re-thrown. To prevent saving without causing the context manager to re-throw an exception, raise `CancelMutation`.

Keyword arguments are passed to the builtin `open` function. Uses `open_with_detected_encoding()` to detect & preserve the encoding. The list of encodings can be overridden by supplying *try_encodings*.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W