

---

**sr.comp**

**unknown**

**Apr 13, 2024**



# CONTENTS

<b>1</b>	<b>User Guide</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	History . . . . .	2
1.3	Compstate Repositories . . . . .	2
1.4	Schedule . . . . .	2
<b>2</b>	<b>API Reference</b>	<b>5</b>
2.1	API . . . . .	5
<b>3</b>	<b>Indices and tables</b>	<b>25</b>
	<b>Python Module Index</b>	<b>27</b>
	<b>Index</b>	<b>29</b>



## **USER GUIDE**

For a guide to using the SRComp suite as a whole, readers are directed to the main [SRComp wiki](#) which address the suite as a whole.

### **1.1 Introduction**

The Student Robotics Competition Software, or *SRComp*, is a suite of software for running competition events. It aims to record the entire state of the competition in a single place and provide tooling for working with that data in a consistent and reproducible manner.

**SRComp assumes:**

- that you have a league section and/or a knockout section; if you have both then the league comes first and seeds the knockout
- that you can generate fair match plan (i.e: who plays who in which match) yourself (though it does provide some tooling to *check* that a plan is fair)

**SRComp includes support for:**

- generating match schedules from match plans, by incorporating both time to reset arenas between matches as well as planned and unexpected delays
- games with multiple participants, with graceful handling of no-shows and disqualifications
- normalising per-game scores to allocate league scores and/or determine knockout progression
- resolving ties
- concurrent arenas, though with the caveat that games in multiple arenas start at the same time and are of the same length
- “shepherds”; people who fetch participants before their matches
- large-screen displays of information for shepherds
- large-screen displays of information for the audience
- web pages with information for an external audience
- web pages with information for competitors
- real-time updates of the state of the competition, including consistent distributed hosting of the displays and HTTP API

## 1.2 History

SRComp was created for [Student Robotics](#)' 2014 competition, and has been used for all subsequent competitions. It has also been used for a number of other similar, though usually smaller, events.

SRComp continues to evolve to support the needs of Student Robotics competitions.

## 1.3 Compstate Repositories

Compstate repositories contain the entire state of the competition at a certain time.

Their directory structure looks something like this:

```
├── arenas.yaml
├── awards.yaml
├── [deployments.yaml]
├── [external]
│   └── [any].yaml
├── league
│   └── [arena]
│       └── [match].yaml
├── knockout
│   └── [arena]
│       └── [match].yaml
├── league.yaml
├── schedule.yaml
├── scoring
│   ├── score.py
│   ├── [converter.py]
│   └── [update.html]
├── shepherding.yaml
└── teams.yaml
```

## 1.4 Schedule

### 1.4.1 Match Slots

Each match is assigned a 'slot' during which it will occur. The times for the slot are generally what is advertised as the match start time, even though the game doesn't actually start until some way into the slot.

### 1.4.2 Match Periods

Matches are grouped into timing periods. Each period has a description, planned start and end times, plus a time beyond which no further matches may be scheduled.

Usually the latter time is after the scheduled end time so that it works to allow for delays to introduce a small overrun if needed. If configured thus, then a period which experiences no delays would end at the scheduled end time.

**Note:** the end times represent the time that the last match in the period can be scheduled to *start* rather than *finish*.

### 1.4.3 Delays

Arbitrary delays can be added to the system at any point. These work to delay the matches that start (currently measured by their slot start) by the given amount, and are cumulative over the course of a period.

### 1.4.4 Staging

Before a match starts each of the teams must submit their robot to the staging area. The system is aware of are various times associated with this:

- The earliest teams can present themselves for a match
- The time by which teams *must* be in staging
- How long staging is open for; equal to the difference between the above
- How long before the start of the match to signal to shepherds they should start looking for teams
- How long before the start of the match to signal to teams they should go to staging





## API REFERENCE

### 2.1 API

#### 2.1.1 Arenas

Arena and corner loading routines.

```
class sr.comp.arenas.Arena(name, display_name, colour)
```

Bases: `NamedTuple`

**colour:** `sr.comp.types.Colour`

Alias for field number 2

**display\_name:** `str`

Alias for field number 1

**name:** `sr.comp.types.ArenaName`

Alias for field number 0

```
class sr.comp.arenas.Corner(number, colour)
```

Bases: `NamedTuple`

**colour:** `sr.comp.types.Colour`

Alias for field number 1

**number:** `sr.comp.arenas.CornerNumber`

Alias for field number 0

```
sr.comp.arenas.load_arenas(filename: pathlib.Path) → dict[sr.comp.types.ArenaName,  
sr.comp.arenas.Arena]
```

Load arenas from a YAML file.

**Parameters** **filename** (*str*) – The filename of the YAML file to load arenas from.

**Returns** A mapping of arena names to [Arena](#) objects.

**Return type** `collections.OrderedDict`

```
sr.comp.arenas.load_corners(filename: pathlib.Path) → dict[sr.comp.arenas.CornerNumber,  
sr.comp.arenas.Corner]
```

Load corner colours from a YAML file.

**Parameters** **filename** (*str*) – The filename of the YAML file to load corners from.

**Returns** A mapping of corner numbers to [Corner](#) objects.

**Return type** `collections.OrderedDict`

## 2.1.2 Competition

Core competition functions.

**class** `sr.comp.comp.SRComp`(*root*: `str` | `pathlib.Path`)

Bases: `object`

A class containing all the various parts of a competition.

**Parameters** `root` (`Path`) – The root path of the compstate repo.

**arenas**

A `collections.OrderedDict` mapping arena names to `sr.comp.arenas.Arena` objects.

**awards**

A `dict` mapping `sr.comp.winners.Award` objects to a `list` of teams.

**corners**

A `collections.OrderedDict` mapping corner numbers to `sr.comp.arenas.Corner` objects.

**schedule**

A `sr.comp.matches.MatchSchedule` instance.

**scores**

A `sr.comp.scores.Scores` instance.

**state**

The current commit of the Compstate repository.

**teams**

A mapping of TLAs to `sr.comp.teams.Team` objects.

**timezone**

The timezone of the competition.

**venue**

A `sr.comp.venue.Venue` instance.

`sr.comp.comp.load_scorer`(*root*: `pathlib.Path`) → `Type[Union[sr.comp.types.ValidatingScorer, sr.comp.types.SimpleScorer]]`

Load the scorer module from Compstate repo.

**Parameters** `root` (`Path`) – The path to the compstate repo.

## 2.1.3 Knockout Schedulers

Knockout schedule generation.

**class** `sr.comp.knockout_scheduler.base_scheduler.BaseKnockoutScheduler`(*schedule*:  
*MatchSchedule*, *scores*:  
*Scores*, *arenas*:  
*Iterable[ArenaName]*,  
*num\_teams\_per\_arena*:  
*int*, *teams*:  
*Mapping[TLA, Team]*,  
*config*: *YAMLData*)

Bases: `object`

Base class for knockout schedulers offering common functionality.

**Parameters**

- **schedule** – The league schedule.
- **scores** – The scores.
- **arenas** (*dict*) – The arenas.
- **teams** (*dict*) – The teams.
- **config** – Custom configuration for the knockout scheduler.

**add\_knockouts()** → *None*

Add the knockouts to the schedule.

Derived classes must override this method.

**static get\_match\_display\_name**(*rounds\_remaining: int, round\_num: int, global\_num: sr.comp.types.MatchNumber*) → *str*

Get a human-readable match display name.

#### Parameters

- **rounds\_remaining** – The number of knockout rounds remaining.
- **knockout\_num** – The match number within the knockout round.
- **global\_num** – The global match number.

**get\_ranking**(*game: sr.comp.match\_period.Match*) → *list[sr.comp.types.TLA]*

Get a ranking of the given match's teams.

**Parameters** **game** – A game.

**num\_teams\_per\_arena**

The number of spaces for teams in an arena.

This is used in building matches where we don't yet know which teams will actually be playing, and for filling in when there aren't enough teams to fill the arena.

**class** *sr.comp.knockout\_scheduler.KnockoutScheduler*(*schedule: MatchSchedule, scores: Scores, arenas: Iterable[ArenaName], num\_teams\_per\_arena: int, teams: Mapping[TLA, Team], config: YAMLData*)

Bases: *sr.comp.knockout\_scheduler.base\_scheduler.BaseKnockoutScheduler*

A class that can be used to generate a knockout schedule based on seeding.

Due to the way the seeding logic works, this class is suitable only when games feature four slots for competitors, with the top two progressing to the next round.

#### Parameters

- **schedule** – The league schedule.
- **scores** – The scores.
- **arenas** (*dict*) – The arenas.
- **num\_teams\_per\_arena** (*int*) – The usual number of teams per arena.
- **teams** (*dict*) – The teams.
- **config** – Custom configuration for the knockout scheduler.

**add\_knockouts()** → *None*

Add the knockouts to the schedule.

Derived classes must override this method.

**static** `get_rounds_remaining(prev_matches: Sized) → int`

**get\_winners**(*game*: `sr.comp.match_period.Match`) → `list[sr.comp.types.TLA]`

Find the parent match's winners.

**Parameters** *game* – A game.

**num\_teams\_per\_arena** = 4

Constant value due to the way the automatic seeding works.

**class** `sr.comp.knockout_scheduler.StaticScheduler(*args: Any, **kwargs: Any)`

Bases: `sr.comp.knockout_scheduler.base_scheduler.BaseKnockoutScheduler`

A knockout scheduler which loads almost fixed data from the config. Assumes only a single arena.

Due to the nature of its interaction with the seedings, this scheduler has a very limited handling of dropped-out teams: it only adjusts its scheduling for dropouts before the knockouts.

**The practical results of this dropout behaviour are:**

- the schedule is stable when teams drop out, as this either affects the entire knockout or none of it
- dropping out a team such that there are no longer enough seeds requires manual changes to the schedule to remove the seeds which cannot be filled

**add\_knockouts**() → `None`

Add the knockouts to the schedule.

Derived classes must override this method.

**get\_team**(*team\_ref*: `Optional[sr.comp.knockout_scheduler.static_scheduler.StaticMatchTeamReference]`) → `Optional[sr.comp.types.TLA]`

## Stable Random

A stable random number generator implementation.

**class** `sr.comp.knockout_scheduler.stable_random.Random`

Bases: `object`

Our own random number generator that is guaranteed to be stable.

Python's random number generator's stability across Python versions is complicated. Different versions will produce different results. It's easier right now to just have our own random number generator that's not as good, but is definitely stable between machines.

---

**Note:** This class is deliberately not a sub-class of `random.Random` since any of the functionality provided by the class (i.e. not just the generation portion) could change between Python versions. Instead, any additionally required functionality should be added below as needed and `_importantly_` tests for the functionality to ensure that the output is the same on all supported platforms.

---

**getrandbits**(*n*: `int`) → `int`

**random**() → `float`

**seed**(*s*: `bytes` | `bytearray` | `memoryview`) → `None`

**shuffle**(*x*: `MutableSequence[sr.comp.knockout_scheduler.stable_random.T]`) → `None`

## 2.1.4 Match Period

Classes that are useful for dealing with match periods.

**class** sr.comp.match\_period.Delay(*delay, time*)

Bases: `NamedTuple`

**delay:** `datetime.timedelta`

Alias for field number 0

**time:** `datetime.datetime`

Alias for field number 1

**class** sr.comp.match\_period.Match(*num, display\_name, arena, teams, start\_time, end\_time, type, use\_resolved\_ranking*)

Bases: `NamedTuple`

**arena:** `sr.comp.types.ArenaName`

Alias for field number 2

**display\_name:** `str`

Alias for field number 1

**end\_time:** `datetime.datetime`

Alias for field number 5

**num:** `sr.comp.types.MatchNumber`

Alias for field number 0

**start\_time:** `datetime.datetime`

Alias for field number 4

**teams:** `list[typing.Optional[sr.comp.types.TLA]]`

Alias for field number 3

**type:** `sr.comp.match_period.MatchType`

Alias for field number 6

**use\_resolved\_ranking:** `bool`

Alias for field number 7

**class** sr.comp.match\_period.MatchPeriod(*start\_time, end\_time, max\_end\_time, description, matches, type*)

Bases: `NamedTuple`

**description:** `str`

Alias for field number 3

**end\_time:** `datetime.datetime`

Alias for field number 1

**matches:** `list[sr.comp.match_period.MatchSlot]`

Alias for field number 4

**max\_end\_time:** `datetime.datetime`

Alias for field number 2

**start\_time:** `datetime.datetime`

Alias for field number 0

**type:** `sr.comp.match_period.MatchType`

Alias for field number 5

```
class sr.comp.match_period.MatchType(value, names=None, *, module=None, qualname=None, type=None,
                                     start=1, boundary=None)
```

Bases: `enum.Enum`

`knockout` = 'knockout'

`league` = 'league'

`tiebreaker` = 'tiebreaker'

## 2.1.5 Match Period Clock

A clock to manage match periods.

```
class sr.comp.match_period_clock.MatchPeriodClock(period: sr.comp.match_period.MatchPeriod,
                                                  delays: Iterable[sr.comp.match_period.Delay])
```

Bases: `object`

A clock for use in scheduling matches within a `MatchPeriod`.

It is generally expected that the time information here will be in the form of `datetime` and `timedelta` instances, though any data which can be compared and added appropriately should work.

Delay rules:

- Only delays which are scheduled after the start of the given period will be considered.
- Delays are cumulative.
- Delays take effect as soon as their `time` is reached.

**advance\_time**(duration: `datetime.timedelta`) → `None`

Make a given amount of time pass. This is expected to be called after scheduling some matches in order to move to the next timeslot.

---

**Note:** It is assumed that the duration value will always be 'positive', i.e. that time will not go backwards. The results of the duration value being 'negative' are undefined.

---

**property current\_time:** `datetime.datetime`

Get the apparent current time. This is a combination of the time which has passed (through calls to `advance_time`) and the delays which have occurred.

Will raise an `OutOfTimeException` if either:

- the end of the period has been reached (i.e: the sum of durations passed to `advance_time` has exceeded the planned duration of the period), or
- the maximum end of the period has been reached (i.e: the current time would be after the period's `max_end_time`).

**static delays\_for\_period**(period: `sr.comp.match_period.MatchPeriod`, delays: `Iterable[sr.comp.match_period.Delay]`) → `list[sr.comp.match_period.Delay]`

Filter and sort a list of all possible delays to include only those which occur after the start of the given `period`.

### Parameters

- **period** (`MatchPeriod`) – The period to get the delays for.
- **delays** (`list`) – The list of `Delays` to consider.

**Returns** A sorted list of delays which occur after the start of the period.

**iterslots**(*slot\_duration: datetime.timedelta*) → Iterator[*datetime.datetime*]

Iterate through all the available timeslots of the given size within the `MatchPeriod`, taking into account delays.

This is equivalent to checking the `current_time` and repeatedly calling `advance_time` with the given duration. As a result, it is safe to call `advance_time` between iterations if additional gaps between slots are needed.

**exception** `sr.comp.match_period_clock.OutOfTimeException`

Bases: `Exception`

An exception representing no more time available at the competition to run matches.

## 2.1.6 Matches

Match schedule library.

**class** `sr.comp.matches.MatchSchedule`(*y: Any, league: sr.comp.types.LeagueMatches, teams: Mapping[sr.comp.types.TLA, sr.comp.teams.Team], num\_teams\_per\_arena: int*)

Bases: `object`

A match schedule.

**add\_tiebreaker**(*scores: sr.comp.scores.Scores, time: datetime.datetime*) → `None`

Add a tie breaker to the league if required. Also set a `tiebreaker` attribute if necessary.

### Parameters

- **scores** (`Scores`) – The scores for the competition.
- **time** (`datetime.datetime`) – The time to have the tiebreaker match.

**classmethod** **create**(*config\_fname: pathlib.Path, league\_fname: pathlib.Path, scores: sr.comp.scores.Scores, arenas: Mapping[sr.comp.types.ArenaName, sr.comp.arenas.Arena], num\_teams\_per\_arena: int, teams: Mapping[sr.comp.types.TLA, sr.comp.teams.Team]*) → `sr.comp.matches.TSchedule`

Create a new match schedule around the given config data.

### Parameters

- **config\_fname** (`Path`) – The filename of the main config file.
- **league\_fname** (`Path`) – The filename of the file containing the league matches.
- **scores** (`Scores`) – The scores for the competition.
- **arenas** (`dict`) – A mapping of arena ids to `Arena` instances.
- **num\_teams\_per\_arena** (`int`) – The usual number of teams per arena.
- **teams** (`dict`) – A mapping of TLAs to `Team` instances.

**property** `datetime_now: datetime.datetime`

Get the current date and time, with the correct timezone.

**delay\_at**(*date: datetime.datetime*) → `datetime.timedelta`

Calculates the active delay at a given date. Intended for use only in exposing the current delay value – scheduling should be done using a `MatchPeriodClock` instead.

**Parameters** **date** (`datetime`) – The date to find the delay for.

**Returns** A `datetime.timedelta` specifying the active delay.

**property final\_match:** *sr.comp.match\_period.Match*

Get the *Match* for the last match of the competition.

This is the info for the ‘finals’ of the competition (i.e: the last of the knockout matches) unless there is a tiebreaker.

**get\_staging\_times**(*match: sr.comp.match\_period.Match*) → *sr.comp.matches.StagingTimes*

**knockout\_rounds:** *list[list[sr.comp.match\_period.Match]]*

A list of the knockout matches by round. Each entry in the list represents a round of knockout matches, such that *knockout\_rounds[-1]* contains a list with only one match – the final.

**match\_periods:** *list[sr.comp.match\_period.MatchPeriod]*

A list of the *MatchPeriod*s which contain the matches for the competition.

**matches:** *list[sr.comp.match\_period.MatchSlot]*

A list of match slots in the schedule. Each match slot is a dict of arena to the *Match* occurring in that arena.

**matches\_at**(*date: datetime.datetime*) → *Iterator[sr.comp.match\_period.Match]*

Get all the matches that occur around a specific date.

**Parameters** *date (datetime)* – The date at which matches occur.

**Returns** An iterable list of matches.

**n\_matches**() → *int*

Get the number of matches.

**Returns** The number of matches.

**n\_planned\_league\_matches**

The number of planned league matches.

**period\_at**(*date: datetime.datetime*) → *sr.comp.match\_period.MatchPeriod | None*

Get the match period that occur around a specific date.

**Parameters** *date (datetime)* – The date at which period occurs.

**Returns** The period at that time or None.

**remove\_drop\_outs**(*teams: Iterable[Optional[sr.comp.types.TLA]], since\_match: sr.comp.types.MatchNumber*) → *list[typing.Optional[sr.comp.types.TLA]]*

Take a list of TLAs and replace the teams that have dropped out with None values.

**Parameters**

- **teams** (*list*) – A list of TLAs.
- **since\_match** (*int*) – The match number to check for drop outs from.

**Returns** A new list containing the appropriate teams.

**teams**

A mapping of TLAs to *Team* instances.

**class** *sr.comp.matches.StagingOffsets*

Bases: *typing\_extensions.TypedDict*

**closes:** *datetime.timedelta*

**duration:** *datetime.timedelta*

**opens:** *datetime.timedelta*

**signal\_shepherds:** *Mapping[sr.comp.types.ShepherdName, datetime.timedelta]*

**signal\_teams:** *datetime.timedelta*



```

class sr.comp.matches.StagingTimes
    Bases: typing_extensions.TypedDict

    closes: datetime.datetime
    duration: datetime.timedelta
    opens: datetime.datetime
    signal_shepherds: Mapping[sr.comp.types.ShepherdName, datetime.datetime]
    signal_teams: datetime.datetime

exception sr.comp.matches.WrongNumberOfTeams(match_n: int, arena_name: str, teams:
    Sequence[Optional[sr.comp.types.TLA]],
    num_teams_per_arena: int)

    Bases: Exception

sr.comp.matches.get_timezone(name: str) → datetime.tzinfo

sr.comp.matches.parse_ranges(ranges: str) → set[int]
    Parse a comma separated list of numbers which may include ranges specified as hyphen-separated numbers.
    From https://stackoverflow.com/questions/6405208

```

## 2.1.7 Raw Compstate

Utilities for working with raw Compstate repositories.

```

class sr.comp.raw_compstate.RawCompstate(path: str | pathlib.Path, local_only: bool)
    Bases: object

```

Helper class to interact with a Compstate as raw files in a Git repository on disk.

### Parameters

- **path** (*Path*) – The path to the Compstate repository.
- **local\_only** (*bool*) – If true, this disabled the pulling, committing and pushing functionality.

```

checkout(what: str) → None

commit(commit_msg: str, allow_empty: bool = False) → None

commit_and_push(commit_msg: str, allow_empty: bool = False) → None

property deployments: list[str]

fetch(where: str = 'origin', refsspecs: Collection[str] = (), quiet: bool = False) → None

get_default_branch() → str

get_score_path(match: sr.comp.match_period.Match) → str
    Get the path to the score file for the given match.

git(command_pieces: Iterable[str], err_msg: str = "", *, return_output: Literal[True]) → str
git(command_pieces: Iterable[str], err_msg: str = "", return_output: Literal[False] = False) → int
git(command_pieces: Iterable[str], err_msg: str = "", return_output: bool = False) → str | int

has_ancestor(commit: str) → bool

property has_changes: bool
    Whether or not there are any changes to files in the state, including untracked files.

```

**has\_commit**(*commit: str*) → bool

Whether or not the given commit is known to this repository.

**has\_descendant**(*commit: str*) → bool

**is\_parent**(*parent: str, child: str*) → bool

**property layout:** **sr.comp.types.LayoutData**

**load**() → *sr.comp.comp.SRComp*

Load the state as an SRComp instance.

**load\_score**(*match: sr.comp.match\_period.Match*) → *sr.comp.types.ScoreData*

Load raw score data for the given match.

**load\_shepherds**() → *list[sr.comp.raw\_compstate.ShepherdInfo]*

Load the shepherds' state.

**pull\_fast\_forward**() → *None*

**push**(*where: str, revspec: str, err\_msg: str = "", force: bool = False*) → *None*

**reset\_and\_fast\_forward**() → *None*

**reset\_hard**() → *None*

**rev\_parse**(*revision: str*) → *str*

**save\_score**(*match: sr.comp.match\_period.Match, score: sr.comp.types.ScoreData*) → *None*

Save raw score data for the given match.

**property shepherding:** **sr.comp.types.ShepherdingData**

Provides access to the raw shepherding data. Most consumers actually want to use `load_shepherds` instead.

**show\_changes**() → *None*

**show\_remotes**() → *None*

**stage**(*file\_path: str*) → *None*

Stage the given file.

**Parameters** **file\_path** (*Path*) – A path to the file to stage. This should either be an absolute path, or one relative to the compstate.

**class** **sr.comp.raw\_compstate.ShepherdInfo**

Bases: *typing\_extensions.TypedDict*

**colour:** **sr.comp.types.Colour**

**name:** **sr.comp.types.ShepherdName**

**regions:** *list[sr.comp.types.RegionName]*

**teams:** *list[sr.comp.types.TLA]*

## 2.1.8 Scores

Utilities for working with scores.

```
class sr.comp.scores.BaseScores(scores_data: Iterable[sr.comp.types.ScoreData], teams:
    Iterable[sr.comp.types.TLA], scorer:
    Type[Union[sr.comp.types.ValidatingScorer, sr.comp.types.SimpleScorer]],
    num_teams_per_arena: int)
```

Bases: `object`

A generic class that holds scores.

### Parameters

- **scores\_data** (*iterable*) – A collection of loaded score sheet data.
- **teams** (*dict*) – The teams in the competition.
- **scorer** (*dict*) – The scorer logic.
- **num\_teams\_per\_arena** (*int*) – The usual number of teams per arena.

```
game_points: dict[typing.Tuple[sr.comp.types.ArenaName, sr.comp.types.MatchNumber],
    typing.Mapping[sr.comp.types.TLA, sr.comp.types.GamePoints]]
```

Game points data for each match. Keys are tuples of the form (arena\_id, match\_num), values are `dicts` mapping TLAs to the number of game points they scored.

```
game_positions: dict[typing.Tuple[sr.comp.types.ArenaName,
    sr.comp.types.MatchNumber], typing.Mapping[league_ranker.RankedPosition,
    set[sr.comp.types.TLA]]]
```

Game position data for each match. Keys are tuples of the form (arena\_id, match\_num), values are `dicts` mapping ranked positions (i.e: first is 1, etc.) to an iterable of TLAs which have that position. Based solely on teams' game points.

```
get_rankings(match: sr.comp.match_period.Match) → Mapping[sr.comp.types.TLA,
    league_ranker.RankedPosition]
```

Return a mapping of TLAs to ranked positions for the given match.

This is an internal API – most consumers should use `Scores.get_scores` instead.

```
property last_scored_match: Optional[sr.comp.types.MatchNumber]
```

The most match with the highest id for which we have score data.

```
ranked_points: dict[typing.Tuple[sr.comp.types.ArenaName,
    sr.comp.types.MatchNumber], dict[sr.comp.types.TLA, league_ranker.LeaguePoints]]
```

Normalised (aka 'league') points earned in each match. Keys are tuples of the form (arena\_id, match\_num), values are `dicts` mapping TLAs to the number of normalised points they would earn for that match.

```
teams: Mapping[sr.comp.types.TLA, sr.comp.scores.TeamScore]
```

Points for each team earned during this portion of the competition. Maps TLAs to `TeamScore` instances.

```
exception sr.comp.scores.DuplicateScoresheet(match_id: Tuple[sr.comp.types.ArenaName,
    sr.comp.types.MatchNumber])
```

Bases: `Exception`

An exception that occurs if two scoresheets for the same match have been entered.

```
exception sr.comp.scores.InvalidTeam(tla: sr.comp.types.TLA, context: str)
```

Bases: `Exception`

An exception that occurs when a score contains an invalid team.

```
class sr.comp.scores.KnockoutScores(scores_data: Iterable[sr.comp.types.ScoreData], teams:
    Iterable[sr.comp.types.TLA], scorer:
    Type[Union[sr.comp.types.ValidatingScorer,
sr.comp.types.SimpleScorer]], num_teams_per_arena: int,
    league_positions: Mapping[sr.comp.types.TLA,
sr.comp.scores.LeaguePosition])
```

Bases: `sr.comp.scores.BaseScores`

A class which holds knockout scores.

```
static calculate_ranking(match_points: Mapping[sr.comp.types.TLA, league_ranker.LeaguePoints],
    league_positions: Mapping[sr.comp.types.TLA,
sr.comp.scores.LeaguePosition]) → dict[sr.comp.types.TLA,
    league_ranker.RankedPosition]
```

Get a ranking of the given match's teams.

#### Parameters

- **match\_points** – A map of TLAs to (normalised) scores.
- **league\_positions** – A map of TLAs to league positions.

```
get_rankings(match: sr.comp.match_period.Match) → Mapping[sr.comp.types.TLA,
    league_ranker.RankedPosition]
```

Return a mapping of TLAs to ranked positions for the given match.

This is an internal API – most consumers should use `Scores.get_scores` instead.

#### resolved\_positions

Position data for each match which includes adjustment for ties. Keys are tuples of the form (arena\_id, match\_num), values are `OrderedDicts` mapping TLAs to the ranked position (i.e: first is 1, etc.) of that team, with the winning team in the start of the list of keys. Tie resolution is done by league position.

```
class sr.comp.scores.LeagueScores(scores_data: Iterable[sr.comp.types.ScoreData], teams:
    Iterable[sr.comp.types.TLA], scorer:
    Type[Union[sr.comp.types.ValidatingScorer,
sr.comp.types.SimpleScorer]], num_teams_per_arena: int, extra:
    Optional[Mapping[sr.comp.types.TLA, sr.comp.scores.TeamScore]] =
    None)
```

Bases: `sr.comp.scores.BaseScores`

A class which holds league scores.

#### positions

An `OrderedDict` of TLAs to `sr.comp.scores.LeaguePositions`.

```
static rank_league(team_scores: Mapping[sr.comp.types.TLA, sr.comp.scores.TeamScore]) →
    Mapping[sr.comp.types.TLA, sr.comp.scores.LeaguePosition]
```

Given a mapping of TLA to `TeamScore`, returns a mapping of TLA to league position which both allows for ties and enables their resolution deterministically.

```
class sr.comp.scores.MatchScore(match_id: 'MatchId', game: 'Mapping[TLA, GamePoints]', normalised:
    'Mapping[TLA, LeaguePoints]', ranking: 'Mapping[TLA,
    RankedPosition]')
```

Bases: `object`

**game:** `Mapping[sr.comp.types.TLA, sr.comp.types.GamePoints]`

**match\_id:** `Tuple[sr.comp.types.ArenaName, sr.comp.types.MatchNumber]`

**normalised:** `Mapping[sr.comp.types.TLA, league_ranker.LeaguePoints]`

```

    ranking: Mapping[sr.comp.types.TLA, league_ranker.RankedPosition]
class sr.comp.scores.Scores(league: sr.comp.scores.LeagueScores, knockout:
    sr.comp.scores.KnockoutScores, tiebreaker: sr.comp.scores.TiebreakerScores)
    Bases: object
    A simple class which stores references to the league and knockout scores.
    get_scores(match: sr.comp.match_period.Match) → sr.comp.scores.MatchScore | None
        Get the scores for a given match.
        Parameters match (sr.comp.match_period.Match) – A match.
        Returns An object describing the scores for the match, if scores have been recorded yet. Other-
            wise None.
        Return type MatchScore | None
    knockout
        The KnockoutScores for the competition.
    last_scored_match
        The match with the highest id for which we have score data.
    league
        The LeagueScores for the competition.
    classmethod load(root: pathlib.Path, teams: Iterable[sr.comp.types.TLA], scorer:
        Type[Union[sr.comp.types.ValidatingScorer, sr.comp.types.SimpleScorer]],
        num_teams_per_arena: int) → sr.comp.scores.Scores
    tiebreaker
        The TiebreakerScores for the competition.
class sr.comp.scores.TeamScore(league: league_ranker.LeaguePoints = 0, game: sr.comp.types.GamePoints
    = 0)
    Bases: object
    A team score.
    Parameters
        • league (int) – The league points.
        • game (int) – The game points.
    add_game_points(score: sr.comp.types.GamePoints) → sr.comp.types.GamePoints
    add_league_points(points: league_ranker.LeaguePoints) → league_ranker.LeaguePoints
class sr.comp.scores.TiebreakerScores(scores_data: Iterable[sr.comp.types.ScoreData], teams:
    Iterable[sr.comp.types.TLA], scorer:
    Type[Union[sr.comp.types.ValidatingScorer,
    sr.comp.types.SimpleScorer]], num_teams_per_arena: int,
    league_positions: Mapping[sr.comp.types.TLA,
    sr.comp.scores.LeaguePosition])
    Bases: sr.comp.scores.KnockoutScores
sr.comp.scores.degroup(grouped_positions: Mapping[sr.comp.scores.T, Iterable[sr.comp.types.TLA]]) →
    collections.OrderedDict[sr.comp.types.TLA, sr.comp.scores.T]
    Given a mapping of positions to collections of teams at that position, returns an OrderedDict of teams to their
    positions.
    Where more than one team has a given position, they are sorted before being inserted.

```

```
sr.comp.scores.get_validated_scores(scorer_cls: Type[Union[sr.comp.types.ValidatingScorer,  
sr.comp.types.SimpleScorer]], input_data: sr.comp.types.ScoreData)  
→ Mapping[sr.comp.types.TLA, sr.comp.types.GamePoints]
```

Helper function which mimics the behaviour from libproton.

Given a libproton 3.0 (Proton 3.0.0-rc2) compatible class this will calculate the scores and validate the input.

```
sr.comp.scores.load_external_scores(scores_data: Iterable[sr.comp.types.ExternalScoreData], teams:  
Iterable[sr.comp.types.TLA]) → Mapping[sr.comp.types.TLA,  
sr.comp.scores.TeamScore]
```

Mechanism to import additional scores from an external source.

This provides flexibility in the sources of score data.

```
sr.comp.scores.load_external_scores_data(result_dir: pathlib.Path) →  
Iterator[sr.comp.types.ExternalScoreData]
```

```
sr.comp.scores.load_scores_data(result_dir: pathlib.Path) → Iterator[sr.comp.types.ScoreData]
```

```
sr.comp.scores.results_finder(root: pathlib.Path) → Iterator[pathlib.Path]
```

An iterator that finds score sheet files.

## 2.1.9 Teams

Team metadata library.

```
class sr.comp.teams.Team(tla, name, rookie, dropped_out_after)
```

Bases: `NamedTuple`

```
dropped_out_after: Optional[sr.comp.types.MatchNumber]
```

Alias for field number 3

```
is_still_around(match_number: sr.comp.types.MatchNumber) → bool
```

Check if this team is still around at a certain match.

**Parameters** `match_number` (`int`) – The number of the match to check.

**Returns** True if the team is still playing.

```
name: str
```

Alias for field number 1

```
rookie: bool
```

Alias for field number 2

```
tla: sr.comp.types.TLA
```

Alias for field number 0

```
sr.comp.teams.load_teams(filename: pathlib.Path) → dict[sr.comp.types.TLA, sr.comp.teams.Team]
```

Load teams from a YAML file.

**Parameters** `filename` (`Path`) – The filename of the YAML file to load.

**Returns** A dictionary mapping TLAs to `Team` objects.

## 2.1.10 Validation

Compstate validation routines.

```
class sr.comp.validation.NaiveValidationError(message: 'str', code: 'str', level: 'ErrorLevel' = 'error')
    Bases: object
    code: str
    level: Literal['error', 'warning', 'hint'] = 'error'
    message: str
    with_source(error_type: sr.comp.validation.ErrorType, id_: object) → sr.comp.validation.ValidationError
```

```
exception sr.comp.validation.ValidationError(message: 'str', code: 'str', source: 'tuple[ErrorType,
    object] | None', level: 'ErrorLevel' = 'error')
```

Bases: `Exception`

code: `str`

level: `Literal['error', 'warning', 'hint'] = 'error'`

message: `str`

source: `tuple[sr.comp.validation.ErrorType, object] | None`

```
sr.comp.validation.find_missing_scores(match_type: sr.comp.match_period.MatchType, match_ids:
    Iterable[Tuple[sr.comp.types.ArenaName,
    sr.comp.types.MatchNumber]], last_match: int | None, schedule:
    Iterable[sr.comp.match_period.MatchSlot]) →
    Sequence[tuple[sr.comp.types.MatchNumber,
    set[sr.comp.types.ArenaName]]]
```

Given a collection of `match_ids` for which we have scores, the `match_type` currently under consideration, the number of the `last_match` which was scored and the list of all known matches determine which scores should be present but aren't.

```
sr.comp.validation.find_teams_without_league_matches(matches:
    Iterable[sr.comp.match_period.MatchSlot],
    possible_teams: Iterable[sr.comp.types.TLA])
    → set[sr.comp.types.TLA]
```

Find teams that don't have league matches.

### Parameters

- **matches** (*list*) – A list of matches.
- **possible\_teams** – A list of possible teams.

**Returns** A `set` of teams without matches.

```
sr.comp.validation.join_and(items: Iterable[object]) → str
```

```
sr.comp.validation.report_errors(error_type: sr.comp.validation.ErrorType, id_: object, errors: list[str])
    → None
```

Print out errors nicely formatted.

### Parameters

- **type** (*str*) – The human-readable 'type'.
- **id** (*str*) – The human-readable 'ID'.
- **errors** (*list*) – A list of string errors.

`sr.comp.validation.report_validation_errors(errors: Sequence[sr.comp.validation.ValidationError]) → None`

`sr.comp.validation.validate(comp: sr.comp.comp.SRComp) → int`  
 Validate a Compstate repo.

**Parameters** `comp` ([sr.comp.SRComp](#)) – A competition instance.

**Returns** The number of errors that have occurred.

`sr.comp.validation.validate_match(match: sr.comp.match\_period.MatchSlot, possible_teams: Iterable[sr.comp.types.TLA]) → Iterator[sr.comp.validation.NaiveValidationError]`

Check that the teams featuring in a match exist and are only required in one arena at a time.

`sr.comp.validation.validate_match_score(match_type: sr.comp.match\_period.MatchType, match_score: Mapping[sr.comp.types.TLA, object], scheduled_match: sr.comp.match\_period.Match) → Iterator[sr.comp.validation.NaiveValidationError]`

Check that the match awards points to the right teams, by checking that the teams with points were scheduled to appear in the match.

`sr.comp.validation.validate_schedule(schedule: sr.comp.matches.MatchSchedule, possible_teams: Iterable[sr.comp.types.TLA], possible_arenas: Container[sr.comp.types.ArenaName]) → Iterator[sr.comp.validation.ValidationError]`

Check that the schedule contains enough time for all the matches, and that the matches themselves are valid.

`sr.comp.validation.validate_schedule_arenas(matches: Iterable[sr.comp.match\_period.MatchSlot], possible_arenas: Container[sr.comp.types.ArenaName]) → Iterator[sr.comp.validation.ValidationError]`

Check that any arena referenced by a match actually exists.

`sr.comp.validation.validate_schedule_count(schedule: sr.comp.matches.MatchSchedule) → Iterator[sr.comp.validation.ValidationError]`

`sr.comp.validation.validate_schedule_timings(scheduled_matches: Iterable[sr.comp.match\_period.MatchSlot], match_duration: datetime.timedelta) → Iterator[sr.comp.validation.ValidationError]`

`sr.comp.validation.validate_scores(match_type: sr.comp.match\_period.MatchType, scores: sr.comp.scores.BaseScores, schedule: Sequence[sr.comp.match\_period.MatchSlot]) → Iterator[sr.comp.validation.ValidationError]`

Validate that the scores are sane.

`sr.comp.validation.validate_scores_inner(match_type: sr.comp.match\_period.MatchType, scores: sr.comp.scores.BaseScores, schedule: Sequence[sr.comp.match\_period.MatchSlot]) → Iterator[sr.comp.validation.ValidationError]`

Validate that scores are sane.

`sr.comp.validation.validate_team_matches(matches: Iterable[sr.comp.match\_period.MatchSlot], possible_teams: Iterable[sr.comp.types.TLA]) → Iterator[sr.comp.validation.ValidationError]`

Check that all teams have been assigned league matches. We don't need (or want) to check the knockouts, since those are scheduled dynamically based on the list of teams.



```
sr.comp.validation.warn_missing_scores(match_type: sr.comp.match_period.MatchType, scores:
    sr.comp.scores.BaseScores, schedule:
    Iterable[sr.comp.match_period.MatchSlot]) →
    Iterator[sr.comp.validation.ValidationError]
```

Check that the scores up to the most recent are all present.

```
sr.comp.validation.with_source(naive_errors: Iterable[sr.comp.validation.NaiveValidationError], source:
    tuple[sr.comp.validation.ErrorType, object]) →
    Iterator[sr.comp.validation.ValidationError]
```

## 2.1.11 Venue

Venue layout metadata library.

```
exception sr.comp.venue.InvalidRegionException(region: sr.comp.types.RegionName, area: str)
    Bases: Exception
```

An exception that occurs when there are invalid regions mentioned in the shepherding data.

```
exception sr.comp.venue.LayoutTeamsException(duplicate_teams: Iterable[sr.comp.types.TLA],
    extra_teams: Iterable[sr.comp.types.TLA],
    missing_teams: Iterable[sr.comp.types.TLA])
    Bases: sr.comp.venue.MismatchException[sr.comp.types.TLA]
```

An exception that occurs when there are duplicate, extra or missing teams in a layout.

```
exception sr.comp.venue.MismatchException(tpl: str, duplicates: Iterable[sr.comp.venue.T_str], extras:
    Iterable[sr.comp.venue.T_str], missing:
    Iterable[sr.comp.venue.T_str])
    Bases: Exception, Generic[sr.comp.venue.T_str]
```

An exception that occurs when there are duplicate, extra or missing items.

```
exception sr.comp.venue.ShepherdingAreasException(where: str, duplicate: Iterable[str], extra:
    Iterable[str], missing: Iterable[str])
    Bases: sr.comp.venue.MismatchException[str]
```

An exception that occurs when there are duplicate, extra or missing shepherding areas in the staging times.

```
class sr.comp.venue.Venue(teams: Iterable[sr.comp.types.TLA], layout_file: pathlib.Path, shepherding_file:
    pathlib.Path)
    Bases: object
```

A class providing information about the layout within the venue.

```
check_staging_times(staging_times: sr.comp.matches.StagingOffsets) → None
```

```
classmethod check_teams(teams: Iterable[sr.comp.types.TLA], teams_layout:
    list[sr.comp.types.RegionData]) → None
```

Check that the given layout of teams contains the same set of teams as the reference.

Will throw a [LayoutTeamsException](#) if there are any missing, extra or duplicate teams found.

### Parameters

- **teams** (*list*) – The reference list of teams in the competition.
- **teams\_layout** (*list*) – A list of maps with a list of teams under the **teams** key.

```
get_team_location(team: sr.comp.types.TLA) → sr.comp.types.RegionName
```

Get the name of the location allocated to the given team within the venue.

**Parameters** **team** (*str*) – The TLA of the team in question.

**Returns** The name of the location allocated to the team.

#### **locations**

A `dict` of location names (from the layout file) to location information, including which teams are in that location and the shepherding region which contains that location.

### 2.1.12 Winners

Calculation of winners of awards.

The awards calculated are:

- 1st place,
- 2nd place,
- 3rd place,
- Rookie award (rookie team with highest league position).

```
class sr.comp.winners.Award(value, names=None, *, module=None, qualname=None, type=None, start=1,
                             boundary=None)
```

Bases: `enum.Enum`

Award types.

These correspond with awards as specified in the rulebook.

```
committee = 'committee'
```

```
first = 'first'
```

```
image = 'image'
```

```
movement = 'movement'
```

```
rookie = 'rookie'
```

```
second = 'second'
```

```
third = 'third'
```

```
web = 'web'
```

```
sr.comp.winners.compute_awards(scores: sr.comp.scores.Scores, final_match: sr.comp.match_period.Match,
                                teams: Mapping[sr.comp.types.TLA, sr.comp.teams.Team], path:
                                pathlib.Path | None = None) → Mapping[sr.comp.winners.Award,
                                List[sr.comp.types.TLA]]
```

Compute the awards handed out from configuration.

#### **Parameters**

- **scores** (`sr.comp.scores.Scores`) – The scores.
- **final\_match** (`Match`) – The match to use as the final.
- **teams** (`dict`) – A mapping from TLAs to `sr.comp.teams.Team` objects.

**Returns** A dictionary of `Award` types to TLAs is returned. This may not have a key for any award type that has not yet been determined.

### 2.1.13 YAML Loader

YAML loading routines.

This includes parsing of dates and times properly, and also ensures the C YAML loader is used which is necessary for optimum performance.

`sr.comp.yaml_loader.add_time_constructor(loader: type[yaml.cyaml.CLoader]) → None`

`sr.comp.yaml_loader.load(file_path: pathlib.Path) → Any`

Load a YAML file and return the results.

**Parameters** `file_path` (*Path*) – The path to the YAML file.

**Returns** The parsed contents.

`sr.comp.yaml_loader.time_constructor(_: Any, node: yaml.nodes.Node) → datetime.datetime`



## INDICES AND TABLES

- `genindex`
- `modindex`
- `search`



## PYTHON MODULE INDEX

### S

- `sr.comp`, 5
- `sr.comp.arenas`, 5
- `sr.comp.comp`, 6
- `sr.comp.knockout_scheduler`, 6
- `sr.comp.knockout_scheduler.stable_random`, 8
- `sr.comp.match_period`, 9
- `sr.comp.match_period_clock`, 10
- `sr.comp.matches`, 11
- `sr.comp.raw_compstate`, 13
- `sr.comp.scores`, 15
- `sr.comp.teams`, 18
- `sr.comp.validation`, 19
- `sr.comp.venue`, 21
- `sr.comp.winners`, 22
- `sr.comp.yaml_loader`, 23





## INDEX

### A

`add_game_points()` (*sr.comp.scores.TeamScore* method), 17  
`add_knockouts()` (*sr.comp.knockout\_scheduler.base\_scheduler.BaseKnockoutScheduler* method), 7  
`add_knockouts()` (*sr.comp.knockout\_scheduler.KnockoutScheduler* method), 7  
`add_knockouts()` (*sr.comp.knockout\_scheduler.StaticScheduler* method), 8  
`add_league_points()` (*sr.comp.scores.TeamScore* method), 17  
`add_tiebreaker()` (*sr.comp.matches.MatchSchedule* method), 11  
`add_time_constructor()` (in module *sr.comp.yaml\_loader*), 23  
`advance_time()` (*sr.comp.match\_period\_clock.MatchPeriodClock* method), 10  
*Arena* (class in *sr.comp.arenas*), 5  
*arena* (*sr.comp.match\_period.Match* attribute), 9  
*arenas* (*sr.comp.comp.SRComp* attribute), 6  
*Award* (class in *sr.comp.winners*), 22  
*awards* (*sr.comp.comp.SRComp* attribute), 6

### B

*BaseKnockoutScheduler* (class in *sr.comp.knockout\_scheduler.base\_scheduler*), 6  
*BaseScores* (class in *sr.comp.scores*), 15

### C

`calculate_ranking()` (*sr.comp.scores.KnockoutScores* static method), 16  
`check_staging_times()` (*sr.comp.venue.Venue* method), 21  
`check_teams()` (*sr.comp.venue.Venue* class method), 21  
`checkout()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
*closes* (*sr.comp.matches.StagingOffsets* attribute), 12  
*closes* (*sr.comp.matches.StagingTimes* attribute), 13  
*code* (*sr.comp.validation.NaiveValidationError* attribute), 19

*code* (*sr.comp.validation.ValidationError* attribute), 19  
*colour* (*sr.comp.arenas.Arena* attribute), 5  
*colour* (*sr.comp.arenas.Corner* attribute), 5  
*colour* (*sr.comp.raw\_compstate.ShepherdInfo* attribute), 14  
`commit()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
`commit_and_push()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
*committee* (*sr.comp.winners.Award* attribute), 22  
`compute_awards()` (in module *sr.comp.winners*), 22  
*Corner* (class in *sr.comp.arenas*), 5  
*corners* (*sr.comp.comp.SRComp* attribute), 6  
`create()` (*sr.comp.matches.MatchSchedule* class method), 11  
*current\_time* (*sr.comp.match\_period\_clock.MatchPeriodClock* property), 10

### D

*datetime\_now* (*sr.comp.matches.MatchSchedule* property), 11  
`degrouper()` (in module *sr.comp.scores*), 17  
*Delay* (class in *sr.comp.match\_period*), 9  
*delay* (*sr.comp.match\_period.Delay* attribute), 9  
`delay_at()` (*sr.comp.matches.MatchSchedule* method), 11  
`delays_for_period()` (*sr.comp.match\_period\_clock.MatchPeriodClock* static method), 10  
*deployments* (*sr.comp.raw\_compstate.RawCompstate* property), 13  
*description* (*sr.comp.match\_period.MatchPeriod* attribute), 9  
*display\_name* (*sr.comp.arenas.Arena* attribute), 5  
*display\_name* (*sr.comp.match\_period.Match* attribute), 9  
*dropped\_out\_after* (*sr.comp.teams.Team* attribute), 18  
*DuplicateScoresheet*, 15  
*duration* (*sr.comp.matches.StagingOffsets* attribute), 12  
*duration* (*sr.comp.matches.StagingTimes* attribute), 13

## E

`end_time` (*sr.comp.match\_period.Match* attribute), 9  
`end_time` (*sr.comp.match\_period.MatchPeriod* attribute), 9

## F

`fetch()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
`final_match` (*sr.comp.matches.MatchSchedule* property), 11  
`find_missing_scores()` (in module *sr.comp.validation*), 19  
`find_teams_without_league_matches()` (in module *sr.comp.validation*), 19  
`first` (*sr.comp.winners.Award* attribute), 22

## G

`game` (*sr.comp.scores.MatchScore* attribute), 16  
`game_points` (*sr.comp.scores.BaseScores* attribute), 15  
`game_positions` (*sr.comp.scores.BaseScores* attribute), 15  
`get_default_branch()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
`get_match_display_name()` (*sr.comp.knockout\_scheduler.base\_scheduler.BaseKnockoutScheduler* static method), 7  
`get_ranking()` (*sr.comp.knockout\_scheduler.base\_scheduler.BaseKnockoutScheduler* method), 7  
`get_rankings()` (*sr.comp.scores.BaseScores* method), 15  
`get_rankings()` (*sr.comp.scores.KnockoutScores* method), 16  
`get_rounds_remaining()` (*sr.comp.knockout\_scheduler.KnockoutScheduler* static method), 7  
`get_score_path()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
`get_scores()` (*sr.comp.scores.Scores* method), 17  
`get_staging_times()` (*sr.comp.matches.MatchSchedule* method), 12  
`get_team()` (*sr.comp.knockout\_scheduler.StaticScheduler* method), 8  
`get_team_location()` (*sr.comp.venue.Venue* method), 21  
`get_timezone()` (in module *sr.comp.matches*), 13  
`get_validated_scores()` (in module *sr.comp.scores*), 17  
`get_winners()` (*sr.comp.knockout\_scheduler.KnockoutScheduler* method), 8  
`getrandbits()` (*sr.comp.knockout\_scheduler.stable\_random.Random* method), 8

`git()` (*sr.comp.raw\_compstate.RawCompstate* method), 13

## H

`has_ancestor()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
`has_changes` (*sr.comp.raw\_compstate.RawCompstate* property), 13  
`has_commit()` (*sr.comp.raw\_compstate.RawCompstate* method), 13  
`has_descendant()` (*sr.comp.raw\_compstate.RawCompstate* method), 14

## I

`image` (*sr.comp.winners.Award* attribute), 22  
`InvalidRegionException`, 21  
`InvalidTeam`, 15  
`is_parent()` (*sr.comp.raw\_compstate.RawCompstate* method), 14  
`is_still_around()` (*sr.comp.teams.Team* method), 18  
`iterslots()` (*sr.comp.match\_period\_clock.MatchPeriodClock* method), 11

## J

`join_and()` (in module *sr.comp.validation*), 19

## K

`knockout` (*sr.comp.matches.MatchSchedule* attribute), 10  
`knockout` (*sr.comp.scores.Scores* attribute), 17  
`knockout_rounds` (*sr.comp.matches.MatchSchedule* attribute), 12  
`KnockoutScheduler` (class in *sr.comp.knockout\_scheduler*), 7  
`KnockoutScores` (class in *sr.comp.scores*), 15

## L

`last_scored_match` (*sr.comp.scores.BaseScores* property), 15  
`last_scored_match` (*sr.comp.scores.Scores* attribute), 17  
`layout` (*sr.comp.raw\_compstate.RawCompstate* property), 14  
`LayoutTeamsException`, 21  
`league` (*sr.comp.match\_period.MatchType* attribute), 10  
`league` (*sr.comp.scores.Scores* attribute), 17  
`LeagueScores` (class in *sr.comp.scores*), 16  
`level` (*sr.comp.validation.NaiveValidationError* attribute), 19  
`level` (*sr.comp.validation.ValidationError* attribute), 19  
`load()` (in module *sr.comp.yaml\_loader*), 23  
`load()` (*sr.comp.raw\_compstate.RawCompstate* method), 14

load() (*sr.comp.scores.Scores class method*), 17  
 load\_arenas() (*in module sr.comp.arenas*), 5  
 load\_corners() (*in module sr.comp.arenas*), 5  
 load\_external\_scores() (*in module sr.comp.scores*), 18  
 load\_external\_scores\_data() (*in module sr.comp.scores*), 18  
 load\_score() (*sr.comp.raw\_compstate.RawCompstate method*), 14  
 load\_scorer() (*in module sr.comp.comp*), 6  
 load\_scores\_data() (*in module sr.comp.scores*), 18  
 load\_shepherds() (*sr.comp.raw\_compstate.RawCompstate method*), 14  
 load\_teams() (*in module sr.comp.teams*), 18  
 locations (*sr.comp.venue.Venue attribute*), 22

## M

Match (*class in sr.comp.match\_period*), 9  
 match\_id (*sr.comp.scores.MatchScore attribute*), 16  
 match\_periods (*sr.comp.matches.MatchSchedule attribute*), 12  
 matches (*sr.comp.match\_period.MatchPeriod attribute*), 9  
 matches (*sr.comp.matches.MatchSchedule attribute*), 12  
 matches\_at() (*sr.comp.matches.MatchSchedule method*), 12  
 MatchPeriod (*class in sr.comp.match\_period*), 9  
 MatchPeriodClock (*class in sr.comp.match\_period\_clock*), 10  
 MatchSchedule (*class in sr.comp.matches*), 11  
 MatchScore (*class in sr.comp.scores*), 16  
 MatchType (*class in sr.comp.match\_period*), 9  
 max\_end\_time (*sr.comp.match\_period.MatchPeriod attribute*), 9  
 message (*sr.comp.validation.NaiveValidationError attribute*), 19  
 message (*sr.comp.validation.ValidationError attribute*), 19  
 MismatchException, 21  
 module  
   sr.comp, 5  
   sr.comp.arenas, 5  
   sr.comp.comp, 6  
   sr.comp.knockout\_scheduler, 6  
   sr.comp.knockout\_scheduler.stable\_random, 8  
   sr.comp.match\_period, 9  
   sr.comp.match\_period\_clock, 10  
   sr.comp.matches, 11  
   sr.comp.raw\_compstate, 13  
   sr.comp.scores, 15  
   sr.comp.teams, 18  
   sr.comp.validation, 19  
   sr.comp.venue, 21

sr.comp.winners, 22  
 sr.comp.yaml\_loader, 23  
 movement (*sr.comp.winners.Award attribute*), 22

## N

n\_matches() (*sr.comp.matches.MatchSchedule method*), 12  
 n\_planned\_league\_matches (*sr.comp.matches.MatchSchedule attribute*), 12  
 NaiveValidationError (*class in sr.comp.validation*), 19  
 name (*sr.comp.arenas.Arena attribute*), 5  
 name (*sr.comp.raw\_compstate.ShepherdInfo attribute*), 14  
 name (*sr.comp.teams.Team attribute*), 18  
 normalised (*sr.comp.scores.MatchScore attribute*), 16  
 num (*sr.comp.match\_period.Match attribute*), 9  
 num\_teams\_per\_arena (*sr.comp.knockout\_scheduler.base\_scheduler.BaseKnockoutScheduler attribute*), 7  
 num\_teams\_per\_arena (*sr.comp.knockout\_scheduler.KnockoutScheduler attribute*), 8  
 number (*sr.comp.arenas.Corner attribute*), 5

## O

opens (*sr.comp.matches.StagingOffsets attribute*), 12  
 opens (*sr.comp.matches.StagingTimes attribute*), 13  
 OutOfTimeException, 11

## P

parse\_ranges() (*in module sr.comp.matches*), 13  
 period\_at() (*sr.comp.matches.MatchSchedule method*), 12  
 positions (*sr.comp.scores.LeagueScores attribute*), 16  
 pull\_fast\_forward() (*sr.comp.raw\_compstate.RawCompstate method*), 14  
 push() (*sr.comp.raw\_compstate.RawCompstate method*), 14

## R

Random (*class in sr.comp.knockout\_scheduler.stable\_random*), 8  
 random() (*sr.comp.knockout\_scheduler.stable\_random.Random method*), 8  
 rank\_league() (*sr.comp.scores.LeagueScores static method*), 16  
 ranked\_points (*sr.comp.scores.BaseScores attribute*), 15  
 ranking (*sr.comp.scores.MatchScore attribute*), 16  
 RawCompstate (*class in sr.comp.raw\_compstate*), 13  
 regions (*sr.comp.raw\_compstate.ShepherdInfo attribute*), 14

remove\_drop\_outs() (*sr.comp.matches.MatchSchedule* method), 12  
 report\_errors() (in module *sr.comp.validation*), 19  
 report\_validation\_errors() (in module *sr.comp.validation*), 19  
 reset\_and\_fast\_forward() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 reset\_hard() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 resolved\_positions (*sr.comp.scores.KnockoutScores* attribute), 16  
 results\_finder() (in module *sr.comp.scores*), 18  
 rev\_parse() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 rookie (*sr.comp.teams.Team* attribute), 18  
 rookie (*sr.comp.winners.Award* attribute), 22  
**S**  
 save\_score() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 schedule (*sr.comp.comp.SRComp* attribute), 6  
 Scores (class in *sr.comp.scores*), 17  
 scores (*sr.comp.comp.SRComp* attribute), 6  
 second (*sr.comp.winners.Award* attribute), 22  
 seed() (*sr.comp.knockout\_scheduler.stable\_random.Random* method), 8  
 ShepherdInfo (class in *sr.comp.raw\_compstate*), 14  
 shepherding (*sr.comp.raw\_compstate.RawCompstate* property), 14  
 ShepherdingAreasException, 21  
 show\_changes() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 show\_remotes() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 shuffle() (*sr.comp.knockout\_scheduler.stable\_random.Random* method), 8  
 signal\_shepherds (*sr.comp.matches.StagingOffsets* attribute), 12  
 signal\_shepherds (*sr.comp.matches.StagingTimes* attribute), 13  
 signal\_teams (*sr.comp.matches.StagingOffsets* attribute), 12  
 signal\_teams (*sr.comp.matches.StagingTimes* attribute), 13  
 source (*sr.comp.validation.ValidationError* attribute), 19  
 sr.comp  
     module, 5  
 sr.comp.arenas  
     module, 5  
 sr.comp.comp  
     module, 6  
 sr.comp.knockout\_scheduler  
     module, 6  
 sr.comp.knockout\_scheduler.stable\_random  
     module, 8  
 sr.comp.match\_period  
     module, 9  
 sr.comp.match\_period\_clock  
     module, 10  
 sr.comp.matches  
     module, 11  
 sr.comp.raw\_compstate  
     module, 13  
 sr.comp.scores  
     module, 15  
 sr.comp.teams  
     module, 18  
 sr.comp.validation  
     module, 19  
 sr.comp.venue  
     module, 21  
 sr.comp.winners  
     module, 22  
 sr.comp.yaml\_loader  
     module, 23  
 SRComp (class in *sr.comp.comp*), 6  
 stage() (*sr.comp.raw\_compstate.RawCompstate* method), 14  
 StagingOffsets (class in *sr.comp.matches*), 12  
 StagingTimes (class in *sr.comp.matches*), 12  
 start\_time (*sr.comp.match\_period.Match* attribute), 9  
 start\_time (*sr.comp.match\_period.MatchPeriod* attribute), 9  
 state (*sr.comp.comp.SRComp* attribute), 6  
 StaticScheduler (class in *sr.comp.knockout\_scheduler*), 8  
**T**  
 Team (class in *sr.comp.teams*), 18  
 teams (*sr.comp.comp.SRComp* attribute), 6  
 teams (*sr.comp.match\_period.Match* attribute), 9  
 teams (*sr.comp.matches.MatchSchedule* attribute), 12  
 teams (*sr.comp.raw\_compstate.ShepherdInfo* attribute), 14  
 teams (*sr.comp.scores.BaseScores* attribute), 15  
 TeamScore (class in *sr.comp.scores*), 17  
 third (*sr.comp.winners.Award* attribute), 22  
 tiebreaker (*sr.comp.match\_period.MatchType* attribute), 10  
 tiebreaker (*sr.comp.scores.Scores* attribute), 17  
 TiebreakerScores (class in *sr.comp.scores*), 17  
 time (*sr.comp.match\_period.Delay* attribute), 9  
 time\_constructor() (in module *sr.comp.yaml\_loader*), 23  
 timezone (*sr.comp.comp.SRComp* attribute), 6  
 tla (*sr.comp.teams.Team* attribute), 18

`type (sr.comp.match_period.Match attribute), 9`  
`type (sr.comp.match_period.MatchPeriod attribute), 9`

## U

`use_resolved_ranking (sr.comp.match_period.Match attribute), 9`

## V

`validate() (in module sr.comp.validation), 20`  
`validate_match() (in module sr.comp.validation), 20`  
`validate_match_score() (in module sr.comp.validation), 20`  
`validate_schedule() (in module sr.comp.validation), 20`  
`validate_schedule_arenas() (in module sr.comp.validation), 20`  
`validate_schedule_count() (in module sr.comp.validation), 20`  
`validate_schedule_timings() (in module sr.comp.validation), 20`  
`validate_scores() (in module sr.comp.validation), 20`  
`validate_scores_inner() (in module sr.comp.validation), 20`  
`validate_team_matches() (in module sr.comp.validation), 20`  
`ValidationError, 19`  
`Venue (class in sr.comp.venue), 21`  
`venue (sr.comp.comp.SRComp attribute), 6`

## W

`warn_missing_scores() (in module sr.comp.validation), 20`  
`web (sr.comp.winners.Award attribute), 22`  
`with_source() (in module sr.comp.validation), 21`  
`with_source() (sr.comp.validation.NaiveValidationError method), 19`  
`WrongNumberOfTeams, 13`