sc2reader Documentation

Release 0.7.0-pre

Graylin Kim

October 07, 2015

Contents

There is a pressing need in the SC2 community for better statistics, better analytics, better tools for organizing and searching replays. Better websites for sharing replays and hosting tournaments. These tools can't be created with out first being able to open up replay files and analyze the content within. That's why **sc2reader** was built, to provide a solid foundation on which the next generation of tools and websites can be built and benefit the community.

So lets get you started right away! Through the linked tutorials and reference pages below we'll get you started building your own tools and systems in no time. Any questions, suggestions, or concerns should be posted to the sc2reader mailing list. You can also pop on to our #sc2reader, our IRC channel on Freenode if you want to chat or need some live support.

Note: Checkout our Frequently Asked Questions. If your question isn't covered there, let us know and we'll add it to the list.

CHAPTER 1

About sc2reader

sc2reader is an open source, MIT licensed, python library for extracting game play information from Starcraft II replay and map files. It is production ready, actively maintained, and hosted publicly on Github [source].

Features:

- Fully parses and extracts all available data from all replay files (arcade included) from every official release (plus the HotS Beta).
- Automatically retrieves maps; extracts basic map data and images. Maps unit type and ability link ids to unit/ability game data.
- Processes replay data into an interlinked set of Team, Player, and Unit objects for easy data manipulation

Plugins:

- Selection Tracking: See every player's current selection and hotkeys at every frame of the game.
- APM Tracking: Provides basic APM information for each player by minute and as game averages.
- GameHeartNormalizer: Fixes teams, races, times, and other oddities typical of GameHeart games.

Scripts:

- sc2printer: Print basic replay information to the terminal.
- sc2json: Render basic replay information to json for use in other languages.
- sc2replayer: Play back a replay one event at a time with detailed printouts.

I am actively looking for community members to assist in documenting the replay data and in creating plugins that enhance functionality. Contact me!

CHAPTER 2

Getting Started

I recommend the following steps when getting started:

- Follow the installation guide
- Read this article on replays: What is in a Replay? (5 minutes).
- Read this article on sc2reader: Concepts in sc2reader (5 minutes).
- Short introduction to sc2reader: Getting Started with SC2Reader (5 minutes)

Now that you've been oriented, you can see sc2reader in action by working through a couple of the tutorials below.

CHAPTER 3

Tutorials

The best way to pick sc2reader up and get started is probably by example. With that in mind, we've written up a series of tutorials on getting various simple tasks done with sc2reader; hopefully they can serve as a quick on ramp for you.

- PrettyPrinter by Example (10-15 minutes)

CHAPTER 4

Articles

A collection of short handwritten articles about aspects of working with replays and sc2reader.

- What is in a Replay? (5 minutes).
- Getting Started with SC2Reader (5 minutes).
- Concepts in sc2reader (5 minutes).
- Creating a GameEngine Plugin (10 minutes).

CHAPTER 5

Reference Pages

Don't forget to check the Frequently Asked Questions if you can't find the answer you are looking for!

5.1 SC2Reader

The replay factory

5.2 Main Structures

The outline of the key structures in the replay object.

5.2.1 Replay

class sc2reader.resources.**Replay**(*replay_file*, *filename=None*, *load_level=4*, *engine=<module 'sc2reader.engine' from '/home/docs/checkouts/readthedocs.org/user_builds/sc2reader/envs/latest/local/lib/python2.7/site-packages/sc2reader-0.7.0rc0-py2.7.egg/sc2reader/engine/__init__.pyc'>*, *do_tracker_events=True*, ***options*)

register_datapack(*datapack*, *filterfunc=<function <lambda>>*)

Allows you to specify your own datapacks for use when loading replays. Datapacks are checked for use with the supplied filterfunc in reverse registration order to give user registered datapacks preference over factory default datapacks.

This is how you would add mappings for your favorite custom map.

Parameters

- **datapack** – A BaseData object to use for mapping unit types and ability codes to their corresponding classes.
- **filterfunc** – A function that accepts a partially loaded *Replay* object as an argument and returns true if the datapack should be used on this replay.

register_default_datapacks()

Registers factory default datapacks.

register_default_readers()

Registers factory default readers.

register_reader(*data_file*, *reader*, *filterfunc=<function <lambda>>*)

Allows you to specify your own reader for use when reading the data files packed into the .SC2Replay archives. Datapacks are checked for use with the supplied filterfunc in reverse registration order to give user registered datapacks preference over factory default datapacks.

Don't use this unless you know what you are doing.

Parameters

- **data_file** – The full file name that you would like this reader to parse.
- **reader** – The `Reader` object you wish to use to read the data file.
- **filterfunc** – A function that accepts a partially loaded *Replay* object as an argument and returns true if the reader should be used on this replay.

5.2.2 Map

class sc2reader.resources.**Map**(*map_file*, *filename=None*, *region=None*, *map_hash=None*, ***options*)

classmethod get_url(*region*, *map_hash*)

Builds a download URL for the map from its components.

5.2.3 Game Summary

class sc2reader.resources.**GameSummary**(*summary_file*, *filename=None*, *lang=u'enUS'*, ***options*)

5.3 Support Structures

These dumb data structures help to give meaningful organization and structure to the information in their respective parent resources.

5.3.1 Entity

class sc2reader.objects.**Entity**(*sid*, *slot_data*)

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity

5.3.2 Player

class sc2reader.objects.**Player**(*pid*, *detail_data*, *attribute_data*)

Parameters

- **pid** (*integer*) – The player's unique player id.
- **detail_data** (*dict*) – The detail data associated with this player
- **attribute_data** (*dict*) – The attribute data associated with this player

5.3.3 User

class sc2reader.objects.**User**(*uid*, *init_data*)

Parameters

- **uid** (*integer*) – The user's unique user id
- **init_data** (*dict*) – The init data associated with this user

url
The player's formatted Battle.net profile url

5.3.4 Observer

class sc2reader.objects.**Observer**(*sid*, *slot_data*, *uid*, *init_data*, *pid*)
Extends *Entity* and *User*.

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity
- **uid** (*integer*) – The user's unique user id
- **init_data** (*dict*) – The init data associated with this user
- **pid** (*integer*) – The player's unique player id.

5.3.5 Computer

class sc2reader.objects.**Computer**(*sid*, *slot_data*, *pid*, *detail_data*, *attribute_data*)
Extends *Entity* and *Player*

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity
- **pid** (*integer*) – The player's unique player id.
- **detail_data** (*dict*) – The detail data associated with this player
- **attribute_data** (*dict*) – The attribute data associated with this player

5.3.6 Participant

class sc2reader.objects.**Participant**(*sid*, *slot_data*, *uid*, *init_data*, *pid*, *detail_data*, *attribute_data*)
Extends *Entity*, *User*, and *Player*

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity
- **uid** (*integer*) – The user's unique user id
- **init_data** (*dict*) – The init data associated with this user

- **pid** (*integer*) – The player's unique player id.
- **detail_data** (*dict*) – The detail data associated with this player
- **attribute_data** (*dict*) – The attribute data associated with this player

5.3.7 Team

class sc2reader.objects.**Team**(*number*)
The team object primarily a container object for organizing *Player* objects with some metadata. As such, it implements iterable and can be looped over like a list.

Parameters **number** (*interger*) – The team number as recorded in the replay

lineup
A string representation of the team play races like PP or TPZZ. Random pick races are not reflected in this string

5.3.8 PlayerSummary

class sc2reader.objects.**PlayerSummary**(*pid*)
Resents a player as loaded from a *GameSummary* file.

5.3.9 Graph

class sc2reader.objects.**Graph**(*x*, *y*, *xy_list=None*)
A class to represent a graph on the score screen. Derived from data in the *GameSummary* file.

as_points()
Get the graph as a list of (x, y) tuples

5.3.10 MapInfo

class sc2reader.objects.**MapInfo**(*contents*)
Represents the data encoded into the MapInfo file inside every SC2Map archive

5.3.11 MapInfoPlayer

class sc2reader.objects.**MapInfoPlayer**(*pid*, *control*, *color*, *race*, *unknown*, *start_point*, *ai*, *decal*)
Describes the player data as found in the MapInfo document of SC2Map archives.

5.4 Data Objects

Objects representing in-game objects.

5.4.1 Unit

class sc2reader.data.**Unit**(*unit_id*)
Represents an in-game unit.

is_army
Boolean flagging units as army units.

is_building
Boolean flagging units as buildings.

is_worker
Boolean flagging units as worker units. SCV, MULE, Drone, Probe

minerals
The mineral cost of the unit. None if no type is assigned

name
The name of the unit type currently active. None if no type is assigned

race
The race of this unit. One of Terran, Protoss, Zerg, Neutral, or None

supply
The supply used by this unit. Negative for supply providers. None if no type is assigned

type
The internal type id of the current unit type of this unit. None if no type is assigned

vespene
The vespene cost of the unit. None if no type is assigned

5.4.2 Ability

class sc2reader.data.**Ability**(*id*, *name=None*, *title=None*, *is_build=False*, *build_time=0*, *build_unit=None*)
Represents an in-game ability

5.4.3 Build

class sc2reader.data.**Build**(*build_id*)

Parameters **build_id** – The build number identifying this dataset.

The datapack for a particualr group of builds. Maps internal integer ids to *Unit* and *Ability* types. Also contains builder methods for creating new units and changing their types.

All build data is valid for standard games only. For arcade maps milage may vary.

change_type(*unit*, *new_type*, *frame*)

Parameters

- **unit** – The changing types.
- **unit_type** – The unit type to assign to this unit

Assigns the given type to a unit.

create_unit(*unit_id*, *unit_type*, *frame*)

Parameters

- **unit_id** – The unique id of this unit.
- **unit_type** – The unit type to assign to the new unit

Creates a new unit and assigns it to the specified type.

5.5 Plugins

sc2reader has a built in game engine that you can plug into to efficiently process replay events. You can add plugins to the engine by calling `register_plugin()`:

```
import sc2reader
from sc2reader.engine.plugins import APMTracker, SelectionTracker
sc2reader.engine.register_plugin(APMTracker())
sc2reader.engine.register_plugin(SelectionTracker())
```

Plugins will be called in order of registration for each event. If plugin B depends on plugin A make sure to register plugin A first!

See the Creating a GameEngine Plugin article for instructions on making your own plugins.

5.5.1 ContextLoader

Note: This plugin is registered by default.

This plugin creates and maintains all the *Unit* and *Ability* data objects from the raw replay data. This creates all the `event.player`, `event.unit`, `event.ability` object references and maintains other game data structures like `objects`.

5.5.2 GameHeartNormalizer

Note: This plugin is registered by default.

This plugin fixes player lists, teams, game lengths, and frames for games that were played with the GameHeart mod.

5.5.3 APMTracker

The `APMTracker` adds three simple fields based on a straight tally of non-camera player action events such as selections, abilities, and hotkeys.

- `player.aps` = a dictionary of second => total actions in that second
- `player.apm` = a dictionary of minute => total actions in that minute
- `player.avg_apm` = Average APM as a float

5.5.4 SelectionTracker

Note: This plugin is intended to be used in conjunction with other user written plugins. If you attempt to use the `player.selection` attribute outside of a registered plugin the values will be the values as they were at the end of the game.

The `SelectionTracker` maintains a `person.selection` structure maps selection buffers for that player to the player's current selection:

```
active_selection = event.player.selection[10]
```

Where buffer is a control group 0-9 or a 10 which represents the active selection.

5.6 Factories

Factories are used to load SCII resources from file-like objects and paths to file-like objects. Objects must implement `read()` such that it retrieves all the file contents.

5.6.1 SC2Factory

class `sc2reader.factories.`**`SC2Factory`**(***options*)

The SC2Factory class acts as a generic loader interface for all available to sc2reader resources. At current time this includes *`Replay`* and *`Map`* resources. These resources can be loaded in both singular and plural contexts with:

- *`load_replay()`* - `Replay`
- *`load_replays()`* - generator<`Replay`>
- *`load_map()`* - `Map`
- *`load_maps()`* - : generator<`Map`>

The load behavior can be configured in three ways:

- Passing options to the factory constructor
- Using the *`configure()`* method of a factory instance
- Passing overried options into the load method

See the *`configure()`* method for more details on configuration options.

Resources can be loaded in the singular context from the following inputs:

- URLs - Uses the built-in package `urllib`
- File path - Uses the built-in method `open`
- File-like object - Must implement `.read()`
- DepotFiles - Describes remote Battle.net depot resources

In the plural context the following inputs are acceptable:

- An iterable of the above inputs
- Directory path - Uses *`get_files()`* with the appropriate extension to fine files.

`configure`(*cls=None*, ***options*)

Configures the factory to use the supplied options. If cls is specified the options will only be applied when loading that class

`load_game_summaries`(*sources*, *options=None*, ***new_options*)

Loads a collection of s2gs files, returns a generator.

load_game_summary(*source*, *options=None*, ***new_options*)
Loads a single s2gs file. Accepts file path, url, or file object.

load_localization(*source*, *options=None*, ***new_options*)
Loads a single s2ml file. Accepts file path, url, or file object.

load_localizations(*sources*, *options=None*, ***new_options*)
Loads a collection of s2ml files, returns a generator.

load_map(*source*, *options=None*, ***new_options*)
Loads a single s2ma file. Accepts file path, url, or file object.

load_maps(*sources*, *options=None*, ***new_options*)
Loads a collection of s2ma files, returns a generator.

load_replay(*source*, *options=None*, ***new_options*)
Loads a single sc2replay file. Accepts file path, url, or file object.

load_replays(*sources*, *options=None*, ***new_options*)
Loads a collection of sc2replay files, returns a generator.

register_plugin(*cls*, *plugin*)
Registers the given Plugin to be run on classes of the supplied name.

reset()
Resets the options to factory defaults

5.6.2 DictCachedSC2Factory

class sc2reader.factories.**DictCachedSC2Factory**(*cache_max_size=0*, ***options*)

Parameters **cache_max_size** – The max number of cache entries to hold in memory.

Extends *SC2Factory*.

Caches remote depot resources in memory. Does not write to the file system. The cache is effectively cleared when the process exits.

5.6.3 FileCachedSC2Factory

class sc2reader.factories.**FileCachedSC2Factory**(*cache_dir*, ***options*)

Parameters **cache_dir** – Local directory to cache files in.

Extends *SC2Factory*.

Caches remote depot resources on the file system in the `cache_dir`.

5.6.4 DoubleCachedSC2Factory

class sc2reader.factories.**DoubleCachedSC2Factory**(*cache_dir*, *cache_max_size=0*, ***options*)

Parameters

- **cache_dir** – Local directory to cache files in.
- **cache_max_size** – The max number of cache entries to hold in memory.

Extends *SC2Factory*.

Caches remote depot resources to the file system AND holds a subset of them in memory for more efficient access.

5.7 Decoders

Used to decode the low level contents of files extracted from MPQFiles

5.7.1 ByteDecoder

class sc2reader.decoders.**ByteDecoder**(*contents*, *endian*)

Parameters

- **contents** – The string or file-like object to decode
- **endian** – Either > or <. Indicates the endian the bytes are stored in.

Used to unpack parse byte aligned files.

done()
Returns true when all bytes have been decoded

peek(*count*)
Returns the raw byte string for the next `count` bytes

read_bytes(*count*)
Returns the next `count` bytes as a byte string

read_cstring(*encoding=u'utf8'*)
Read a NULL byte terminated character string decoded with given encoding (default utf8). Ignores endian.

read_range(*start*, *end*)
Returns the raw byte string from the indicated address range

read_string(*count*, *encoding=u'utf8'*)
Read a string in given encoding (default utf8) that is `count` bytes long

read_uint(*count*)
Returns the next `count` bytes as an unsigned integer

read_uint16()
Returns the next two bytes as an unsigned integer

read_uint32()
Returns the next four bytes as an unsigned integer

read_uint64()
Returns the next eight bytes as an unsigned integer

read_uint8()
Returns the next byte as an unsigned integer

5.7.2 BitPackedDecoder

class sc2reader.decoders.**BitPackedDecoder**(*contents*)

Parameters **contents** – The string of file-like object to decode

Extends *ByteDecoder*. Always packed BIG_ENDIAN

Adds capabilities for parsing files that Blizzard has packed in bits and not in bytes.

byte_align()
Moves cursor to the beginning of the next byte

done()
Returns true when all bytes in the buffer have been used

read_aligned_bytes(*count*)
Skips to the beginning of the next byte and returns the next `count` bytes as a byte string

read_aligned_string(*count*, *encoding=u'utf8'*)
Skips to the beginning of the next byte and returns the next `count` bytes decoded with encoding (default utf8)

read_bits(*count*)
Returns the next `count` bits as an unsigned integer

read_bytes(*count*)
Returns the next `count*8` bits as a byte string

read_frames()
Reads a frame count as an unsigned integer

read_struct(*datatype=None*)
Reads a nested data structure. If the type is not specified the first byte is used as the type identifier.

read_uint16()
Returns the next 16 bits as an unsigned integer

read_uint32()
Returns the next 32 bits as an unsigned integer

read_uint64()
Returns the next 64 bits as an unsigned integer

read_uint8()
Returns the next 8 bits as an unsigned integer

read_vint()
Reads a signed integer of variable length

5.8 Utilities

These utilities are provided to make working with certain types of data a bit easier.

5.8.1 DepotFile

class sc2reader.utils.**DepotFile**(*bytes*)

Parameters **bytes** – The raw bytes representing the depot file

The DepotFile object parses bytes for a dependency into their components and assembles them into a URL so that the dependency can be fetched.

url
Returns url of the depot file.

5.8.2 Color

class sc2reader.utils.**Color**(*name=None*, *r=0*, *g=0*, *b=0*, *a=255*)

Stores a color name and rgba representation of a color. Individual color components can be retrieved with the dot operator:

```
color = Color(r=255, g=0, b=0, a=75)
tuple(color.r,color.g, color.b, color.a) == color.rgba
```

You can also create a color by name.

color = Color('Red')

Only standard Starcraft colors are supported. ValueErrors will be thrown on invalid names or hex values.

hex

The hexadecimal representation of the color

rgba

Returns a tuple containing the color's (r,g,b,a)

5.8.3 Length

class sc2reader.utils.**Length**

Extends the builtin timedelta class. See python docs for more info on what capabilities this gives you.

hours

The number of hours in represented.

mins

The number of minutes in excess of the hours.

secs

The number of seconds in excess of the minutes.

5.8.4 PersonDict

5.8.5 AttributeDict

5.8.6 get_files

sc2reader.utils.**get_files**(*path*, *exclude=[]*, *depth=-1*, *followlinks=False*, *extension=None*, ***extras*)

Retrieves files from the given path with configurable behavior.

Parameters

- **path** – Path to search for files
- **depth** – Limits the depth of the search. -1 = Unlimited
- **followLinks** – Enables the search to follow links.
- **exclude** – Excludes subdirectories with names in this list.
- **extension** – Restricts results to files matching the given extension."

5.9 Events

All of the gamplay and state information contained in the replay is packed into events.

- Game Events: Human actions and certain triggered events
- Message Events: Message and Pings to other players.
- Tracker Events: Game state information

5.9.1 Game Events

Game events are what the Starcraft II engine uses to reconstruct games for you to watch and take over in. Because the game is deterministic, only event data directly created by a player action is recorded. These player actions are then replayed automatically when watching a replay. Because the AI is 100% deterministic no events are ever recorded for a computer player.

class sc2reader.events.game.**AddToControlGroupEvent** (*frame*, *pid*, *data*)
Extends *ControlGroupEvent*

This event adds the current selection to the control group.

class sc2reader.events.game.**BasicCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

This event is recorded for events that have no extra information recorded.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**CameraEvent** (*frame*, *pid*, *data*)
Camera events are generated when ever the player camera moves, zooms, or rotates. It does not matter why the camera changed, this event simply records the current state of the camera after changing.

class sc2reader.events.game.**CommandEvent** (*frame*, *pid*, *data*)
Ability events are generated when ever a player in the game issues a command to a unit or group of units. They are split into three subclasses of ability, each with their own set of associated data. The attributes listed below are shared across all ability event types.

See *TargetPointCommandEvent*, *TargetUnitCommandEvent*, and *DataCommandEvent* for individual details.

class sc2reader.events.game.**ControlGroupEvent** (*frame*, *pid*, *data*)
ControlGroup events are recorded when ever a player action modifies or accesses a control group. There are three kinds of events, generated by each of the possible player actions:

- `SetControlGroup` - Recorded when a user sets a control group (ctrl+#).
- `GetControlGroup` - Recorded when a user retrieves a control group (#).
- `AddToControlGroup` - Recorded when a user adds to a control group (shift+ctrl+#)

All three events have the same set of data (shown below) but are interpretted differently. See the class entry for details.

class sc2reader.events.game.**DataCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

DataCommandEvent are recorded when ever a player issues a command that has no target. Commands like Burrow, SeigeMode, Train XYZ, and Stop fall under this category.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**GameEvent** (*frame*, *pid*)
This is the base class for all game events. The attributes below are universally available.

class sc2reader.events.game.**GameStartEvent** (*frame*, *pid*, *data*)
Recorded when the game starts and the frames start to roll. This is a global non-player event.

class sc2reader.events.game.**GetControlGroupEvent** (*frame*, *pid*, *data*)
Extends *ControlGroupEvent*

This event replaces the current selection with the contents of the control group. The mask data is used to limit that selection to units that are currently selectable. You might have 1 medivac and 8 marines on the control group but if the 8 marines are inside the medivac they cannot be part of your selection.

class sc2reader.events.game.**HijackReplayGameEvent** (*frame*, *pid*, *data*)
Generated when players take over from a replay.

class sc2reader.events.game.**PlayerLeaveEvent** (*frame*, *pid*, *data*)
Recorded when a player leaves the game.

class sc2reader.events.game.**ResourceRequestCancelEvent** (*frame*, *pid*, *data*)
Generated when a player cancels their resource request.

class sc2reader.events.game.**ResourceRequestEvent** (*frame*, *pid*, *data*)
Generated when a player creates a resource request.

class sc2reader.events.game.**ResourceRequestFulfillEvent** (*frame*, *pid*, *data*)
Generated when a player accepts a resource request.

class sc2reader.events.game.**ResourceTradeEvent** (*frame*, *pid*, *data*)
Generated when a player trades resources with another player. But not when fullfulling resource requests.

class sc2reader.events.game.**SelectionEvent** (*frame*, *pid*, *data*)
Selection events are generated when ever the active selection of the player is updated. Unlike other game events, these events can also be generated by non-player actions like unit deaths or transformations.

Starting in Starcraft 2.0.0, selection events targetting control group buffers are also generated when control group selections are modified by non-player actions. When a player action updates a control group a *ControlGroupEvent* is generated.

class sc2reader.events.game.**SetControlGroupEvent** (*frame*, *pid*, *data*)
Extends *ControlGroupEvent*

This event does a straight forward replace of the current control group contents with the player's current selection. This event doesn't have masks set.

class sc2reader.events.game.**TargetPointCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

This event is recorded when ever a player issues a command that targets a location and NOT a unit. Commands like Psistorm, Attack Move, Fungal Growth, and EMP fall under this category.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**TargetUnitCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

This event is recorded when ever a player issues a command that targets a unit. The location of the target unit at the time of the command is also recorded. Commands like Chronoboost, Transfuse, and Snipe fall under this category.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**UserOptionsEvent** (*frame*, *pid*, *data*)
This event is recorded for each player at the very beginning of the game before the *GameStartEvent*.

5.9.2 Message Events

class sc2reader.events.message.**ChatEvent** (*frame*, *pid*, *target*, *text*)
Records in-game chat events.

class sc2reader.events.message.**MessageEvent** (*frame*, *pid*)
Parent class for all message events.

class sc2reader.events.message.**PingEvent** (*frame*, *pid*, *target*, *x*, *y*)
Records pings made by players in game.

class sc2reader.events.message.**ProgressEvent** (*frame*, *pid*, *progress*)
Sent during the load screen to update load process for other clients.

5.9.3 Tracker Events

Tracker events are new in Starcraft patch 2.0.8. These events are generated by the game engine when important non-player events occur in the game. Some of them are also periodically recorded to snapshot aspects of the current game state.

class sc2reader.events.tracker.**PlayerSetupEvent** (*frames*, *data*, *build*)
Sent during game setup to help us organize players better

class sc2reader.events.tracker.**PlayerStatsEvent** (*frames*, *data*, *build*)
Player Stats events are generated for all players that were in the game even if they've since left every 10 seconds. An additional set of stats events are generated at the end of the game.

When a player leaves the game, a single PlayerStatsEvent is generated for that player and no one else. That player continues to generate PlayerStatsEvents at 10 second intervals until the end of the game.

In 1v1 games, the above behavior can cause the losing player to have 2 events generated at the end of the game. One for leaving and one for the end of the game.

class sc2reader.events.tracker.**TrackerEvent** (*frames*)
Parent class for all tracker events.

class sc2reader.events.tracker.**UnitBornEvent** (*frames*, *data*, *build*)
Generated when a unit is created in a finished state in the game. Examples include the Marine, Zergling, and Zealot (when trained from a gateway). Units that enter the game unfinished (all buildings, warped in units) generate a *UnitInitEvent* instead.

Unfortunately, units that are born do not have events marking their beginnings like *UnitInitEvent* and *UnitDoneEvent* do. The closest thing to it are the CommandEvent game events where the command is a train unit command.

class sc2reader.events.tracker.**UnitDiedEvent** (*frames*, *data*, *build*)
Generated when a unit dies or is removed from the game for any reason. Reasons include morphing, merging, and getting killed.

class sc2reader.events.tracker.**UnitDoneEvent** (*frames*, *data*, *build*)
The counter part to the *UnitInitEvent*, generated by the game engine when an initiated unit is completed. E.g. warp-in finished, building finished, morph complete.

class sc2reader.events.tracker.**UnitInitEvent** (*frames*, *data*, *build*)
The counter part to *UnitDoneEvent*, generated by the game engine when a unit is initiated. This applies only to units which are started in game before they are finished. Primary examples being buildings and warp-in units.

class sc2reader.events.tracker.**UnitOwnerChangeEvent** (*frames*, *data*, *build*)
Generated when either ownership or control of a unit is changed. Neural Parasite is an example of an action that would generate this event.

class sc2reader.events.tracker.**UnitPositionsEvent** (*frames*, *data*, *build*)
Generated every 15 seconds. Marks the positions of the first 255 units that were damaged in the last interval. If more than 255 units were damaged, then the first 255 are reported and the remaining units are carried into the next interval.

class sc2reader.events.tracker.**UnitTypeChangeEvent** (*frames*, *data*, *build*)
Generated when the unit's type changes. This generally tracks upgrades to buildings (Hatch, Lair, Hive) and mode switches (Sieging Tanks, Phasing prisms, Burrowing roaches). There may be some other situations where a unit transformation is a type change and not a new unit.

class sc2reader.events.tracker.**UpgradeCompleteEvent** (*frames*, *data*, *build*)
Generated when a player completes an upgrade.

5.10 Frequently Asked Questions

1. *How do I get a list of game events (including messages)?*
2. *How to I get the game state at a specific time in the game. E.g. at 10:00 how many workers every player has.*
3. *How can I retrieve game summary files?*
4. *Script <name here> is broken. What is wrong?*

5.10.1 How do I get a list of game events (including messages)?

Here is a minimal example:

```
replay = sc2reader.load_replay('path/to/replay.SC2Replay')
for event in replay.events:
    print '{0} => {1}: {2}'.format(event.pid,event.name, event.time)
```

Please see the documentation for a full listing of the information available.

5.10.2 How to I get the game state at a specific time in the game. E.g. at 10:00 how many workers every player has.

This is difficult. Events are only recorded for player initiated actions and you'll find that both successful and unsuccessful actions are included. That means several things complicates our lives:

1. There is no "unit created" event. Only a "player attempted to use train <unit>" events.
2. There is no "death" event. You can only tell a unit is alive when it is actively selected.
3. Game state information: player resources, available supply, etc are unavailable at all times.

It may be possible to overcome these limitations and approximate game state with a series of very smart assumptions and cool algorithms. If you could accurately count workers though, you'd be the first I think.

5.10.3 How can I retrieve game summary files?

s2gs files hashes are not contained inside any other SC2 resources as far as anyone knows.

Make sure you read the s2gs thread for details.

Aside from manually causing s2gs files to download to your battle.net cache folder you might try set up the S2GSExtractor to scrape them from the process memory. I can't speak to its legality or effectiveness but it is somewhere to start if you want to automate things.

5.10.4 Script <name here> is broken. What is wrong?

It is true that not all the scripts are very well maintained. They were originally intended as mini usage examples. It seems that people are trying to use them as a primary interface for sc2reader though. I'll have to make sure they don't break going forward.

Patches to the scripts are always accepted, just issue a pull request or email me a patch file.

5.11 SC2Reader

The replay factory

5.12 Main Structures

The outline of the key structures in the replay object.

5.12.1 Replay

class sc2reader.resources.**Replay**(*replay_file*, *filename=None*, *load_level=4*, *engine=<module 'sc2reader.engine' from '/home/docs/checkouts/readthedocs.org/user_builds/sc2reader/envs/latest/local/lib/python2.7/site-packages/sc2reader-0.7.0rc0-py2.7.egg/sc2reader/engine/__init__.pyc'>*, *do_tracker_events=True*, ***options*)

register_datapack(*datapack*, *filterfunc=<function <lambda>>*)

Allows you to specify your own datapacks for use when loading replays. Datapacks are checked for use with the supplied filterfunc in reverse registration order to give user registered datapacks preference over factory default datapacks.

This is how you would add mappings for your favorite custom map.

Parameters

- **datapack** – A BaseData object to use for mapping unit types and ability codes to their corresponding classes.
- **filterfunc** – A function that accepts a partially loaded *Replay* object as an argument and returns true if the datapack should be used on this replay.

register_default_datapacks()

Registers factory default datapacks.

register_default_readers()

Registers factory default readers.

register_reader(*data_file*, *reader*, *filterfunc=<function <lambda>>*)

Allows you to specify your own reader for use when reading the data files packed into the .SC2Replay archives. Datapacks are checked for use with the supplied filterfunc in reverse registration order to give user registered datapacks preference over factory default datapacks.

Don't use this unless you know what you are doing.

Parameters

- **data_file** – The full file name that you would like this reader to parse.
- **reader** – The `Reader` object you wish to use to read the data file.
- **filterfunc** – A function that accepts a partially loaded *Replay* object as an argument and returns true if the reader should be used on this replay.

5.12.2 Map

class sc2reader.resources.**Map**(*map_file*, *filename=None*, *region=None*, *map_hash=None*, ***options*)

classmethod get_url(*region*, *map_hash*)

Builds a download URL for the map from its components.

5.12.3 Game Summary

class sc2reader.resources.**GameSummary**(*summary_file*, *filename=None*, *lang=u'enUS'*, ***options*)

5.13 Support Structures

These dumb data structures help to give meaningful organization and structure to the information in their respective parent resources.

5.13.1 Entity

class sc2reader.objects.**Entity**(*sid*, *slot_data*)

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity

5.13.2 Player

class sc2reader.objects.**Player**(*pid*, *detail_data*, *attribute_data*)

Parameters

- **pid** (*integer*) – The player's unique player id.
- **detail_data** (*dict*) – The detail data associated with this player
- **attribute_data** (*dict*) – The attribute data associated with this player

5.13.3 User

class sc2reader.objects.**User**(*uid*, *init_data*)

Parameters

- **uid** (*integer*) – The user's unique user id
- **init_data** (*dict*) – The init data associated with this user

url
The player's formatted Battle.net profile url

5.13.4 Observer

class sc2reader.objects.**Observer**(*sid*, *slot_data*, *uid*, *init_data*, *pid*)
Extends *Entity* and *User*.

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity
- **uid** (*integer*) – The user's unique user id
- **init_data** (*dict*) – The init data associated with this user
- **pid** (*integer*) – The player's unique player id.

5.13.5 Computer

class sc2reader.objects.**Computer**(*sid*, *slot_data*, *pid*, *detail_data*, *attribute_data*)
Extends *Entity* and *Player*

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity
- **pid** (*integer*) – The player's unique player id.
- **detail_data** (*dict*) – The detail data associated with this player
- **attribute_data** (*dict*) – The attribute data associated with this player

5.13.6 Participant

class sc2reader.objects.**Participant**(*sid*, *slot_data*, *uid*, *init_data*, *pid*, *detail_data*, *attribute_data*)
Extends *Entity*, *User*, and *Player*

Parameters

- **sid** (*integer*) – The entity's unique slot id.
- **slot_data** (*dict*) – The slot data associated with this entity
- **uid** (*integer*) – The user's unique user id
- **init_data** (*dict*) – The init data associated with this user

- **pid** (*integer*) – The player's unique player id.
- **detail_data** (*dict*) – The detail data associated with this player
- **attribute_data** (*dict*) – The attribute data associated with this player

5.13.7 Team

class sc2reader.objects.**Team**(*number*)
The team object primarily a container object for organizing *Player* objects with some metadata. As such, it implements iterable and can be looped over like a list.

Parameters **number** (*interger*) – The team number as recorded in the replay

lineup
A string representation of the team play races like PP or TPZZ. Random pick races are not reflected in this string

5.13.8 PlayerSummary

class sc2reader.objects.**PlayerSummary**(*pid*)
Resents a player as loaded from a *GameSummary* file.

5.13.9 Graph

class sc2reader.objects.**Graph**(*x*, *y*, *xy_list=None*)
A class to represent a graph on the score screen. Derived from data in the *GameSummary* file.

as_points()
Get the graph as a list of (x, y) tuples

5.13.10 MapInfo

class sc2reader.objects.**MapInfo**(*contents*)
Represents the data encoded into the MapInfo file inside every SC2Map archive

5.13.11 MapInfoPlayer

class sc2reader.objects.**MapInfoPlayer**(*pid*, *control*, *color*, *race*, *unknown*, *start_point*, *ai*, *decal*)
Describes the player data as found in the MapInfo document of SC2Map archives.

5.14 Data Objects

Objects representing in-game objects.

5.14.1 Unit

class sc2reader.data.**Unit**(*unit_id*)
Represents an in-game unit.

is_army
Boolean flagging units as army units.

is_building
Boolean flagging units as buildings.

is_worker
Boolean flagging units as worker units. SCV, MULE, Drone, Probe

minerals
The mineral cost of the unit. None if no type is assigned

name
The name of the unit type currently active. None if no type is assigned

race
The race of this unit. One of Terran, Protoss, Zerg, Neutral, or None

supply
The supply used by this unit. Negative for supply providers. None if no type is assigned

type
The internal type id of the current unit type of this unit. None if no type is assigned

vespene
The vespene cost of the unit. None if no type is assigned

5.14.2 Ability

class sc2reader.data.**Ability**(*id*, *name=None*, *title=None*, *is_build=False*, *build_time=0*, *build_unit=None*)
Represents an in-game ability

5.14.3 Build

class sc2reader.data.**Build**(*build_id*)

Parameters **build_id** – The build number identifying this dataset.

The datapack for a particualr group of builds. Maps internal integer ids to *Unit* and *Ability* types. Also contains builder methods for creating new units and changing their types.

All build data is valid for standard games only. For arcade maps milage may vary.

change_type(*unit*, *new_type*, *frame*)

Parameters

- **unit** – The changing types.
- **unit_type** – The unit type to assign to this unit

Assigns the given type to a unit.

create_unit(*unit_id*, *unit_type*, *frame*)

Parameters

- **unit_id** – The unique id of this unit.
- **unit_type** – The unit type to assign to the new unit

Creates a new unit and assigns it to the specified type.

5.15 Plugins

sc2reader has a built in game engine that you can plug into to efficiently process replay events. You can add plugins to the engine by calling `register_plugin()`:

```
import sc2reader
from sc2reader.engine.plugins import APMTracker, SelectionTracker
sc2reader.engine.register_plugin(APMTracker())
sc2reader.engine.register_plugin(SelectionTracker())
```

Plugins will be called in order of registration for each event. If plugin B depends on plugin A make sure to register plugin A first!

See the Creating a GameEngine Plugin article for instructions on making your own plugins.

5.15.1 ContextLoader

Note: This plugin is registered by default.

This plugin creates and maintains all the *Unit* and *Ability* data objects from the raw replay data. This creates all the `event.player`, `event.unit`, `event.ability` object references and maintains other game data structures like `objects`.

5.15.2 GameHeartNormalizer

Note: This plugin is registered by default.

This plugin fixes player lists, teams, game lengths, and frames for games that were played with the GameHeart mod.

5.15.3 APMTracker

The `APMTracker` adds three simple fields based on a straight tally of non-camera player action events such as selections, abilities, and hotkeys.

- `player.aps` = a dictionary of second => total actions in that second
- `player.apm` = a dictionary of minute => total actions in that minute
- `player.avg_apm` = Average APM as a float

5.15.4 SelectionTracker

Note: This plugin is intended to be used in conjunction with other user written plugins. If you attempt to use the `player.selection` attribute outside of a registered plugin the values will be the values as they were at the end of the game.

The `SelectionTracker` maintains a `person.selection` structure maps selection buffers for that player to the player's current selection:

```
active_selection = event.player.selection[10]
```

Where buffer is a control group 0-9 or a 10 which represents the active selection.

5.16 Factories

Factories are used to load SCII resources from file-like objects and paths to file-like objects. Objects must implement `read()` such that it retrieves all the file contents.

5.16.1 SC2Factory

class `sc2reader.factories.`**`SC2Factory`**(***options*)

The SC2Factory class acts as a generic loader interface for all available to sc2reader resources. At current time this includes *`Replay`* and *`Map`* resources. These resources can be loaded in both singular and plural contexts with:

- *`load_replay()`* - `Replay`
- *`load_replays()`* - generator<`Replay`>
- *`load_map()`* - `Map`
- *`load_maps()`* - : generator<`Map`>

The load behavior can be configured in three ways:

- Passing options to the factory constructor
- Using the *`configure()`* method of a factory instance
- Passing overried options into the load method

See the *`configure()`* method for more details on configuration options.

Resources can be loaded in the singular context from the following inputs:

- URLs - Uses the built-in package `urllib`
- File path - Uses the built-in method `open`
- File-like object - Must implement `.read()`
- DepotFiles - Describes remote Battle.net depot resources

In the plural context the following inputs are acceptable:

- An iterable of the above inputs
- Directory path - Uses *`get_files()`* with the appropriate extension to fine files.

`configure`(*cls=None*, ***options*)

Configures the factory to use the supplied options. If cls is specified the options will only be applied when loading that class

`load_game_summaries`(*sources*, *options=None*, ***new_options*)

Loads a collection of s2gs files, returns a generator.

load_game_summary(*source*, *options=None*, ***new_options*)
Loads a single s2gs file. Accepts file path, url, or file object.

load_localization(*source*, *options=None*, ***new_options*)
Loads a single s2ml file. Accepts file path, url, or file object.

load_localizations(*sources*, *options=None*, ***new_options*)
Loads a collection of s2ml files, returns a generator.

load_map(*source*, *options=None*, ***new_options*)
Loads a single s2ma file. Accepts file path, url, or file object.

load_maps(*sources*, *options=None*, ***new_options*)
Loads a collection of s2ma files, returns a generator.

load_replay(*source*, *options=None*, ***new_options*)
Loads a single sc2replay file. Accepts file path, url, or file object.

load_replays(*sources*, *options=None*, ***new_options*)
Loads a collection of sc2replay files, returns a generator.

register_plugin(*cls*, *plugin*)
Registers the given Plugin to be run on classes of the supplied name.

reset()
Resets the options to factory defaults

5.16.2 DictCachedSC2Factory

class sc2reader.factories.**DictCachedSC2Factory**(*cache_max_size=0*, ***options*)

Parameters **cache_max_size** – The max number of cache entries to hold in memory.

Extends *SC2Factory*.

Caches remote depot resources in memory. Does not write to the file system. The cache is effectively cleared when the process exits.

5.16.3 FileCachedSC2Factory

class sc2reader.factories.**FileCachedSC2Factory**(*cache_dir*, ***options*)

Parameters **cache_dir** – Local directory to cache files in.

Extends *SC2Factory*.

Caches remote depot resources on the file system in the `cache_dir`.

5.16.4 DoubleCachedSC2Factory

class sc2reader.factories.**DoubleCachedSC2Factory**(*cache_dir*, *cache_max_size=0*, ***options*)

Parameters

- **cache_dir** – Local directory to cache files in.
- **cache_max_size** – The max number of cache entries to hold in memory.

Extends *SC2Factory*.

Caches remote depot resources to the file system AND holds a subset of them in memory for more efficient access.

5.17 Decoders

Used to decode the low level contents of files extracted from MPQFiles

5.17.1 ByteDecoder

class sc2reader.decoders.**ByteDecoder**(*contents*, *endian*)

Parameters

- **contents** – The string or file-like object to decode
- **endian** – Either > or <. Indicates the endian the bytes are stored in.

Used to unpack parse byte aligned files.

done()
Returns true when all bytes have been decoded

peek(*count*)
Returns the raw byte string for the next `count` bytes

read_bytes(*count*)
Returns the next `count` bytes as a byte string

read_cstring(*encoding=u'utf8'*)
Read a NULL byte terminated character string decoded with given encoding (default utf8). Ignores endian.

read_range(*start*, *end*)
Returns the raw byte string from the indicated address range

read_string(*count*, *encoding=u'utf8'*)
Read a string in given encoding (default utf8) that is `count` bytes long

read_uint(*count*)
Returns the next `count` bytes as an unsigned integer

read_uint16()
Returns the next two bytes as an unsigned integer

read_uint32()
Returns the next four bytes as an unsigned integer

read_uint64()
Returns the next eight bytes as an unsigned integer

read_uint8()
Returns the next byte as an unsigned integer

5.17.2 BitPackedDecoder

class sc2reader.decoders.**BitPackedDecoder**(*contents*)

Parameters **contents** – The string of file-like object to decode

Extends *ByteDecoder*. Always packed BIG_ENDIAN

Adds capabilities for parsing files that Blizzard has packed in bits and not in bytes.

byte_align()
Moves cursor to the beginning of the next byte

done()
Returns true when all bytes in the buffer have been used

read_aligned_bytes(*count*)
Skips to the beginning of the next byte and returns the next `count` bytes as a byte string

read_aligned_string(*count*, *encoding=u'utf8'*)
Skips to the beginning of the next byte and returns the next `count` bytes decoded with encoding (default utf8)

read_bits(*count*)
Returns the next `count` bits as an unsigned integer

read_bytes(*count*)
Returns the next `count*8` bits as a byte string

read_frames()
Reads a frame count as an unsigned integer

read_struct(*datatype=None*)
Reads a nested data structure. If the type is not specified the first byte is used as the type identifier.

read_uint16()
Returns the next 16 bits as an unsigned integer

read_uint32()
Returns the next 32 bits as an unsigned integer

read_uint64()
Returns the next 64 bits as an unsigned integer

read_uint8()
Returns the next 8 bits as an unsigned integer

read_vint()
Reads a signed integer of variable length

5.18 Utilities

These utilities are provided to make working with certain types of data a bit easier.

5.18.1 DepotFile

class sc2reader.utils.**DepotFile**(*bytes*)

Parameters **bytes** – The raw bytes representing the depot file

The DepotFile object parses bytes for a dependency into their components and assembles them into a URL so that the dependency can be fetched.

url
Returns url of the depot file.

5.18.2 Color

class sc2reader.utils.**Color**(*name=None*, *r=0*, *g=0*, *b=0*, *a=255*)

Stores a color name and rgba representation of a color. Individual color components can be retrieved with the dot operator:

```
color = Color(r=255, g=0, b=0, a=75)
tuple(color.r,color.g, color.b, color.a) == color.rgba
```

You can also create a color by name.

color = Color('Red')

Only standard Starcraft colors are supported. ValueErrors will be thrown on invalid names or hex values.

hex

The hexadecimal representation of the color

rgba

Returns a tuple containing the color's (r,g,b,a)

5.18.3 Length

class sc2reader.utils.**Length**

Extends the builtin timedelta class. See python docs for more info on what capabilities this gives you.

hours

The number of hours in represented.

mins

The number of minutes in excess of the hours.

secs

The number of seconds in excess of the minutes.

5.18.4 PersonDict

5.18.5 AttributeDict

5.18.6 get_files

sc2reader.utils.**get_files**(*path*, *exclude=[]*, *depth=-1*, *followlinks=False*, *extension=None*, ***extras*)

Retrieves files from the given path with configurable behavior.

Parameters

- **path** – Path to search for files
- **depth** – Limits the depth of the search. -1 = Unlimited
- **followLinks** – Enables the search to follow links.
- **exclude** – Excludes subdirectories with names in this list.
- **extension** – Restricts results to files matching the given extension."

5.19 Adding new Datapacks

TODO

5.20 Concepts in sc2reader

Some of the important concepts in sc2reader in no particular order.

5.20.1 Factories

All resources are loaded through a factory. There are four kinds:

- *SC2Factory* - Basic factory. Loads resources.
- *DictCachedSC2Factory* - Caches remote resources in memory. When loading remote resources, the dict cache is checked first.
- *FileCachedSC2Factory* - Caches remote resources on the file system. When loading remote resources, the file system is checked first.
- *DoubleCachedSC2Factory* - Caches remote resource in memory and on the file system.

A default factory is automatically configured and attached to the `sc2reader` module when the library is imported. Calling any factory method on the sc2reader module will use this default factory:

```
sc2reader.configure(debug=True)
replay = sc2reader.load_replay('my_replay.SC2Replay')
```

The default factory can be configured with the following environment variables:

- SC2READER_CACHE_DIR - Enables caching to file at the specified directory.
- SC2READER_CACHE_MAX_SIZE - Enables memory caching of resources with a maximum number of entries; not based on memory imprint!

5.20.2 Resources

A Starcraft II resource is any Starcraft II game file. This primarily refers to *Replay* and *Map* resources but also includes less common resources such as *GameSummary*, `Localization`, and SC2Mods.

5.20.3 Player vs User

All entities in a replay fall into one of two overlapping buckets:

- User: A human entity, only users have game and message events.
- Player: A entity that actively plays in the game, only players have tracker events.

As such the following statements are true:

- A Participant is a Player **and** a User
- An Observer is a User **and not** a Player
- An Computer is a Player **and not** a User

5.20.4 Game vs Real

Many attributes in sc2reader are prefixed with `game_` and `real_`. Game refers to the value encoded in the replay. Real refers to the real life value, as best as we can tell. For instance, `game_type` might be 2v2 but by looking at the teams we know that `real_type` is 1v1.

5.20.5 GameEngine

The game engine is used to process replay events and augument the replay with new statistics and game state. It implements a plugin system that allows developers to inject their own logic into the game loop. It also allows plugins to `yield` new events to the event stream. This allows for basic message passing between plugins.

A default engine is automatically configured and attached to the `sc2reader.engine` module when the library is imported. Calling any game engine method on the engine module will use this default factory:

```
sc2reader.engine.register_plugin(MyPlugin())
sc2reader.engine.run(replay)
```

5.20.6 Datapack

A datapack is a collection of *`Unit`* and *`Ability`* classes that represent the game meta data for a given replay. Because this information is not stored in a replay, sc2reader ships with a datapack for standard ladder games from each Starcraft patch.

For non-standard maps, this datapack will be both wrong and incomplete and the Unit/Ability data should not be trusted. If you want to add a datapack for your map, see the article on Adding new Datapacks.

5.21 Creating a GameEngine Plugin

5.21.1 Handling Events

Plugins can opt in to handle events with methods with the following naming convention:

```
def handleEventName(self, event, replay)
```

In addition to handling specific event types, plugins can also handle events more generally by handling built-in parent classes from the list below:

- handleEvent - called for every single event of all types
- handleMessageEvent - called for events in replay.message.events
- handleGameEvent - called for events in replay.game.events
- handleTrackerEvent - called for events in replay.tracker.events
- handlePlayerActionEvent - called for all game events indicating player actions
- handleAbilityEvent - called for all types of ability events
- handleHotkeyEvent - called for all player hotkey events

For every event in a replay, the GameEngine will loop over all of its registered plugins looking for functions to handle that event. Matching handlers are called in order of plugin registration from most general to most specific.

Given the following plugins:

```
class Plugin1():
    def handleAbilityEvent(self, event, replay):
        pass

class Plugin2():
    def handleEvent(self, event, replay):
        pass

    def handleTargetAbilityEvent(self, event, replay):
        pass

sc2reader.engine.register_plugin(Plugin1())
sc2reader.engine.register_plugin(Plugin2())
```

When the engine handles a `TargetAbilityEvent` it will call handlers in the following order:

```
Plugin1.handleAbilityEvent(event, replay)
Plugin2.handleEvent(event, replay)
Plugin2.handleTargetAbilityEvent(event, replay)
```

5.21.2 Setup and Cleanup

Plugins may also handle special `InitGame` and `EndGame` events. These handlers for these events are called directly before and after the processing of the replay events:

- handleInitGame - is called prior to processing a new replay to provide an opportunity for the plugin to clear internal state and set up any replay state necessary.
- handleEndGame - is called after all events have been processed and can be used to perform post processing on aggrated data or clean up intermediate data caches.

5.21.3 Message Passing

Event handlers can choose to `yield` additional events which will be injected into the event stream directly after the event currently being processed. This feature allows for message passing between plugins. An ExpansionTracker plugin could notify all other plugins of a new ExpansionEvent that they could opt to process:

```
def handleUnitDoneEvent(self, event, replay):
        if event.unit.name == 'Nexus':
                yield ExpansionEvent(event.frame, event.unit)
        ...
```

5.21.4 Early Exits

If a plugin wishes to stop processing a replay it can yield a PluginExit event before returning:

```
def handleEvent(self, event, replay):
        if len(replay.tracker_events) == 0:
                yield PluginExit(self, code=0, details=dict(msg="tracker events required"))
                return
        ...

def handleAbilityEvent(self, event, replay):
        try:
                possibly_throwing_error()
```

```
        catch Error as e:
                logger.error(e)
                yield PluginExit(self, code=0, details=dict(msg="Unexpected exception"))
                return
```

The GameEngine will intercept this event and remove the plugin from the list of active plugins for this replay. The exit code and details will be available from the replay:

```
code, details = replay.plugins['MyPlugin']
```

5.21.5 Using Your Plugin

To use your plugin with sc2reader, just register it to the game engine:

sc2reader.engine.register_plugin(MyPlugin())

Plugins will be called in order of registration for each event. If plugin B depends on plugin A make sure to register plugin A first!

5.22 Getting Started with SC2Reader

5.22.1 Loading Replays

For many users, the most basic commands will handle all of their needs:

```
import sc2reader
replay = sc2reader.load_replay('MyReplay', load_map=true)
```

This will load all replay data and fix GameHeart games. In some cases, you don't need the full extent of the replay data. You can use the load level option to limit replay loading and improve load times:

```
# Release version and game length info. Nothing else
sc2reader.load_replay('MyReplay.SC2Replay', load_level=0)

# Also loads game details: map, speed, time played, etc
sc2reader.load_replay('MyReplay.SC2Replay', load_level=1)

# Also loads players and chat events:
sc2reader.load_replay('MyReplay.SC2Replay', load_level=2)

# Also loads tracker events:
sc2reader.load_replay('MyReplay.SC2Replay', load_level=3)

# Also loads game events:
sc2reader.load_replay('MyReplay.SC2Replay', load_level=4)
```

If you want to load a collection of replays, you can use the plural form. Loading resources in this way returns a replay generator:

```
replays = sc2reader.load_replays('path/to/replay/directory')
```

5.22.2 Loading Maps

If you have a replay and want the map file as well, sc2reader can download the corresponding map file and load it in one of two ways:

```
replay = sc2reader.load_replay('MyReplay.SC2Replay', load_map=true)
replay.load_map()
```

If you are looking to only handle maps you can use the map specific load methods:

```
map = sc2reader.load_map('MyMap.SC2Map')
map = sc2reader.load_maps('path/to/maps/directory')
```

5.22.3 Using the Cache

If you are loading a lot of remote resources, you'll want to enable caching for sc2reader. Caching can be configured with the following environment variables:

- SC2READER_CACHE_DIR - Enables caching to file at the specified directory.
- SC2READER_CACHE_MAX_SIZE - Enables memory caching of resources with a maximum number of entries; not based on memory imprint!

You can set these from inside your script with the following code **BEFORE** importing the sc2reader module:

```
os.environ['SC2READER_CACHE_DIR'] = "path/to/local/cache"
os.environ['SC2READER_CACHE_MAX_SIZE'] = 100

# if you have imported sc2reader anywhere already this won't work
import sc2reader
```

5.22.4 Using Plugins

There are a growing number of community generated plugins that you can take advantage of in your project. See the article on Creating a GameEngine Plugin for details on creating your own. To use these plugins you need to customize the game engine:

```
from sc2reader.engine.plugins import SelectionTracker, APMTracker
sc2reader.engine.register_plugin(SelectionTracker())
sc2reader.engine.register_plugin(APMTracker())
```

The `ContextLoader` and `GameHeartNormalizer` plugins are registered by default.

5.23 What is in a Replay?

A SC2Replay file is an archive, just like a like zip or tar file. Inside there are several files, each with a specific purpose. The important ones can be split into three categories.

5.23.1 Initial State:

These first three files describe the initial state of the game before any events occur.

- replay.initData - Records game client information and lobby slot data.

- replay.details - Records basic player and game data.
- replay.attributes.events - Records assorted player and game attributes from the lobby.

The Starcraft II game client can be thought of as a deterministic state machine. Given an initial state and a list of events, the end state can be exactly replicated.

5.23.2 Input Events:

The next two files provide a feed of player actions in the game.

- replay.message.events - Records chat messages and pings.
- replay.game.events - Records every action of every person in the game.

When you watch a replay the game just reads from these feeds. When you take over from a replay, the game client cuts the feeds and switches over to live mouse/keyboard input. Because the AI is deterministic the replay never contains message/game events for them.

5.23.3 Output Events:

The last file provides a record of important events from the game.

- replay.tracker.events - Records important game events and game state updates.

This file was introduced in 2.0.4 and is unncessary for the Starcraft II to reproduce the game. Instead, it records interesting game events and game state for community developers to use when analyzing replays.

5.23.4 What isn't in a replay?

Replays are specifically designed to only include data essential to recreate the game. Game state is not recorded because the game engine can recreate it based off the other information. That means no player resource counts, colleciton rates, supply values, vision, unit positions, unit deaths, etc. Information that you are super interested in probably is not directly recorded. Fortunately since 2.0.4 tracker events now record some of this information; prior to that patch we had to run our own simulations to guess at most of the data.

The other important aspect of this is that instead of completely describing all of the game data (unit data, ability data, map info, etc), replays maintain a list of dependencies. These dependencies might look like this:

- Core.SC2Mod
- Liberty (multi).SC2Mod
- Swarm (multi).SC2Mod
- Teams2.SC2Mod
- Current Patch.SC2mod
- GameHeart.SC2Mod
- Map.SC2Map

As part of the replay pre-load process, each of these dependencies is fetched and all of their associated data is loaded into memory. When Battle.net tells you it is "Fetching Files", it can often be referring to dependencies like this.

sc2reader has attempted to mitigate this serious deficiency by packaging its own game data files. Basic cost, build time, and race information has been packaged. For many people this will be enough. Future versions of sc2reader will provide support for game data exports from the World Editor (introduced in patch 2.0.10). These exports should provide a much more robust dataset to work with.

5.24 PrettyPrinter by Example

To walk through the sc2reader's basic interfaces we're going to step through the process of writing a pretty printer that will work like this:

```
$ python prettyPrinter.py "test_replays/1.4.0.19679/The Boneyard (10).SC2Replay"

test_replays/1.4.0.19679/The Boneyard (10).SC2Replay
------------------------------------------
SC2 Version 1.4.0.19679
Ladder Game, 2011-09-20 21:08:08
2v2 on The Boneyard
Length:17.12

Team 1   (Z) Remedy
         (Z) ShadesofGray

Team 2   (P) KingTroy
         (R) Jamir
```

5.24.1 Getting Started

The first step is to get a script up to accept a path from the command line.

```
import sys

def main():
    path = sys.argv[1]

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

Now we need to use sc2reader to read that file into a *Replay* object that contains all the information that we are looking for.

```
from sc2reader.factories import SC2Factory

def main():
    path = sys.argv[1]
    sc2 = SC2Factory()
    replay = sc2.load_replay(path)
```

In the code above, we imported the *SC2Factory* class from the `sc2reader.factories` package. This class is a factory class that is used to load replays. This factory is configurable in a variety of ways but sane defaults are provided so that most users probably won't need to do any substantial configuration. In fact, because many users will never need to configure the SC2Factory the package provides a default factory that can be used by operating directly on the sc2reader package.

```
import sc2reader

def main():
    path = sys.argv[1]
    replay = sc2reader.load_replay(path)
```

We'll use this short hand method for the rest of this tutorial.

The replay object itself is a dumb data structure; there are no access methods only attributes. For our script we will need the following set of attributes, a full list of attributes can be found on the *Replay* reference page.

```
>>> replay.filename
'test_replays/1.4.0.19679/The Boneyard (10).SC2Replay'
>>> replay.release_string
'1.4.0.19679'
>>> replay.category
'Ladder'
>>> replay.end_time
datetime.datetime(2011, 9, 20, 21, 8, 8)
>>> replay.type
'2v2'
>>> replay.map_name
'The Boneyard'
>>> replay.game_length # string format is MM.SS
Length(0, 1032)
>>> replay.teams
[<sc2reader.objects.Team object at 0x2b5e410>, <sc2reader.objects.Team object at 0x2b5e4d0>]
```

Many of the replay attributes are nested data structures which are generally all pretty dumb as well. The idea being that you should be able to access almost anything with a mixture of indexes and dot notation. Clean and simple accessibility is one of the primary design goals for sc2reader.

```
>>> replay.teams[0].players[0].color.hex
'B4141E'
>>> replay.player.name('Remedy').url
'http://us.battle.net/sc2/en/profile/2198663/1/Remedy/'
```

Each of these nested structures can be found either on its own reference page or lumped together with the other minor structures on the Misc Structures page.

So now all we need to do is build the ouput using the available replay attributes. Lets start with the header portion. We'll use a block string formatting method that makes this clean and easy:

```
def formatReplay(replay):
    return """

{filename}
-----------------------------------------
SC2 Version {release_string}
{category} Game, {start_time}
{type} on {map_name}
Length: {game_length}

""".format(**replay.__dict__)
```

In the code above we are taking advantage of the dump data structure design of the *Replay* objects and unpacking its internal dictionary into the `string.format` method. If you aren't familiar with some of these concepts they are well worth reading up on; string templates are awesome!

Similar formatting written in a more verbose and less pythonic way:

```
def formatReplay(replay):
    output  = replay.filename+'\n'
    output += "-----------------------------------------\n"
    output += "SC2 Version "+replay.release_string+'\n'
    output += replay.category+" Game, "+str(replay.start_time)+'\n'
    output += replay.type+" on "+replay.map_name+'\n'
```

```
    output += "Length: "+str(replay.game_length)
    return output
```

Next we need to format the teams for display. The *Team* and *Player* data structures are pretty straightforward as well so we can apply a bit of string formatting and produce this:

```
def formatTeams(replay):
    teams = list()
    for team in replay.teams:
        players = list()
        for player in team:
            players.append("({0}) {1}".format(player.pick_race[0], player.name))
        formattedPlayers = '\n          '.join(players)
        teams.append("Team {0}:  {1}".format(team.number, formattedPlayers))
    return '\n\n'.join(teams)
```

So lets put it all together into the final script, `prettyPrinter.py`:

```
import sys

import sc2reader

def formatTeams(replay):
    teams = list()
    for team in replay.teams:
        players = list()
        for player in team:
            players.append("({0}) {1}".format(player.pick_race[0], player.name))
        formattedPlayers = '\n          '.join(players)
        teams.append("Team {0}:  {1}".format(team.number, formattedPlayers))
    return '\n\n'.join(teams)

def formatReplay(replay):
    return """
{filename}
----------------------------------------
SC2 Version {release_string}
{category} Game, {start_time}
{type} on {map_name}
Length: {game_length}

{formattedTeams}
""".format(formattedTeams=formatTeams(replay), **replay.__dict__)


def main():
    path = sys.argv[1]
    replay = sc2reader.load_replay(path)
    print formatReplay(replay)

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

5.24.2 Making Improvements

So our script works fine for single files, but what if you want to handle multiple files or directories? sc2reader provides two functions for loading replays: *load_replay()* and *load_replays()* which return a single replay and a

list respectively. *load_replay()* was used above for convenience but *load_replays()* is much more versitile. Here's the difference:

- *load_replay()*: accepts a file path or an opened file object.
- *load_replays()*: accepts a collection of opened file objects or file paths. Can also accept a single path to a directory; files will be pulled from the directory using *get_files()* and the given options.

With this in mind, lets make a slight change to our main function to support any number of paths to any combination of files and directories:

```
def main():
    paths = sys.argv[1:]
    for replay in sc2reader.load_replays(paths):
        print formatReplay(replay)
```

Any time that you start dealing with directories or collections of files you run into dangers with recursion and annoyances of tedium. sc2reader provides options to mitigate these concerns.

- directory: Default ''. The directory string when supplied, becomes the base of all the file paths sent into sc2reader and can save you the hassle of fully qualifying your file paths each time.
- depth: Default -1. When handling directory inputs, sc2reader searches the directory recursively until all .SC2Replay files have been loaded. By setting the maxium depth value this behavior can be mitigated.
- exclude: Default []. When recursing directories you can choose to exclude directories from the file search by directory name (not full path).
- followlinks: Default false. When recursing directories, enables or disables the follow symlinks behavior.

Remember above that the short hand notation that we have been using works with a default instance of the SC2Factory class. *sc2reader.factories.SC2Factory* objects can be customized either on construction or with the *configure()* method.

```
from sc2reader.factories import SC2Factory

sc2 = SC2Factory(
    directory='~/Documents/Starcraft II/Multiplayer/Replays',
    exclude=['Customs','Pros'],
    followlinks=True
)

sc2.configure(depth=1)
```

Recognizing that most users will only ever need one active configuration at a time, the default factory the package makes available is configurable as well.

```
import sc2reader

sc2reader.configure(
    directory='~/Documents/Starcraft II/Multiplayer/Replays',
    exclude=['Customs','Pros'],
    depth=1,
    followlinks=True
)
```

Many of your replay files might be custom games for which events cannot be parsed. Since the pretty printer doesn't use game event information in its output we can limit the parse level of the replay to stop after loading the basic details. This will make the pretty printer work for custom games as well.

```
import sc2reader

sc2reader.load_replay(path, load_level=1)
```

There are 4 available load levels:

- 0: Parses the replay header for version, build, and length information
- 1: Also parses the replay.details, replay.attribute.events and replay.initData files for game settings, map, and time information
- 2: Also parses the replay.message.events file and constructs game teams and players.
- 3: Also parses the replay.tracker.events file
- 4: Also parses the replay.game.events file for player action events.

So that's it! An ideal prettyPrinter script might let the user configure these options as arguments using the `argparse` library. Such an expansion is beyond the scope of sc2reader though, so we'll leave it that one to you.

5.25 Events

All of the gamplay and state information contained in the replay is packed into events.

- Game Events: Human actions and certain triggered events
- Message Events: Message and Pings to other players.
- Tracker Events: Game state information

5.25.1 Game Events

Game events are what the Starcraft II engine uses to reconstruct games for you to watch and take over in. Because the game is deterministic, only event data directly created by a player action is recorded. These player actions are then replayed automatically when watching a replay. Because the AI is 100% deterministic no events are ever recorded for a computer player.

class `sc2reader.events.game.`**`AddToControlGroupEvent`**(*frame*, *pid*, *data*)
Extends *`ControlGroupEvent`*

This event adds the current selection to the control group.

class `sc2reader.events.game.`**`BasicCommandEvent`**(*frame*, *pid*, *data*)
Extends *`CommandEvent`*

This event is recorded for events that have no extra information recorded.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class `sc2reader.events.game.`**`CameraEvent`**(*frame*, *pid*, *data*)
Camera events are generated when ever the player camera moves, zooms, or rotates. It does not matter why the camera changed, this event simply records the current state of the camera after changing.

class `sc2reader.events.game.`**`CommandEvent`**(*frame*, *pid*, *data*)
Ability events are generated when ever a player in the game issues a command to a unit or group of units. They are split into three subclasses of ability, each with their own set of associated data. The attributes listed below are shared across all ability event types.

See *TargetPointCommandEvent*, *TargetUnitCommandEvent*, and *DataCommandEvent* for individual details.

class sc2reader.events.game.**ControlGroupEvent** (*frame*, *pid*, *data*)
ControlGroup events are recorded when ever a player action modifies or accesses a control group. There are three kinds of events, generated by each of the possible player actions:

- `SetControlGroup` - Recorded when a user sets a control group (ctrl+#).
- `GetControlGroup` - Recorded when a user retrieves a control group (#).
- `AddToControlGroup` - Recorded when a user adds to a control group (shift+ctrl+#)

All three events have the same set of data (shown below) but are interpretted differently. See the class entry for details.

class sc2reader.events.game.**DataCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

DataCommandEvent are recorded when ever a player issues a command that has no target. Commands like Burrow, SeigeMode, Train XYZ, and Stop fall under this category.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**GameEvent** (*frame*, *pid*)
This is the base class for all game events. The attributes below are universally available.

class sc2reader.events.game.**GameStartEvent** (*frame*, *pid*, *data*)
Recorded when the game starts and the frames start to roll. This is a global non-player event.

class sc2reader.events.game.**GetControlGroupEvent** (*frame*, *pid*, *data*)
Extends *ControlGroupEvent*

This event replaces the current selection with the contents of the control group. The mask data is used to limit that selection to units that are currently selectable. You might have 1 medivac and 8 marines on the control group but if the 8 marines are inside the medivac they cannot be part of your selection.

class sc2reader.events.game.**HijackReplayGameEvent** (*frame*, *pid*, *data*)
Generated when players take over from a replay.

class sc2reader.events.game.**PlayerLeaveEvent** (*frame*, *pid*, *data*)
Recorded when a player leaves the game.

class sc2reader.events.game.**ResourceRequestCancelEvent** (*frame*, *pid*, *data*)
Generated when a player cancels their resource request.

class sc2reader.events.game.**ResourceRequestEvent** (*frame*, *pid*, *data*)
Generated when a player creates a resource request.

class sc2reader.events.game.**ResourceRequestFulfillEvent** (*frame*, *pid*, *data*)
Generated when a player accepts a resource request.

class sc2reader.events.game.**ResourceTradeEvent** (*frame*, *pid*, *data*)
Generated when a player trades resources with another player. But not when fullfulling resource requests.

class sc2reader.events.game.**SelectionEvent** (*frame*, *pid*, *data*)
Selection events are generated when ever the active selection of the player is updated. Unlike other game events, these events can also be generated by non-player actions like unit deaths or transformations.

Starting in Starcraft 2.0.0, selection events targetting control group buffers are also generated when control group selections are modified by non-player actions. When a player action updates a control group a *ControlGroupEvent* is generated.

class sc2reader.events.game.**SetControlGroupEvent** (*frame*, *pid*, *data*)
Extends *ControlGroupEvent*

This event does a straight forward replace of the current control group contents with the player's current selection. This event doesn't have masks set.

class sc2reader.events.game.**TargetPointCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

This event is recorded when ever a player issues a command that targets a location and NOT a unit. Commands like Psistorm, Attack Move, Fungal Growth, and EMP fall under this category.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**TargetUnitCommandEvent** (*frame*, *pid*, *data*)
Extends *CommandEvent*

This event is recorded when ever a player issues a command that targets a unit. The location of the target unit at the time of the command is also recorded. Commands like Chronoboost, Transfuse, and Snipe fall under this category.

Note that like all CommandEvents, the event will be recorded regardless of whether or not the command was successful.

class sc2reader.events.game.**UserOptionsEvent** (*frame*, *pid*, *data*)
This event is recorded for each player at the very beginning of the game before the *GameStartEvent*.

5.25.2 Message Events

class sc2reader.events.message.**ChatEvent** (*frame*, *pid*, *target*, *text*)
Records in-game chat events.

class sc2reader.events.message.**MessageEvent** (*frame*, *pid*)
Parent class for all message events.

class sc2reader.events.message.**PingEvent** (*frame*, *pid*, *target*, *x*, *y*)
Records pings made by players in game.

class sc2reader.events.message.**ProgressEvent** (*frame*, *pid*, *progress*)
Sent during the load screen to update load process for other clients.

5.25.3 Tracker Events

Tracker events are new in Starcraft patch 2.0.8. These events are generated by the game engine when important non-player events occur in the game. Some of them are also periodically recorded to snapshot aspects of the current game state.

class sc2reader.events.tracker.**PlayerSetupEvent** (*frames*, *data*, *build*)
Sent during game setup to help us organize players better

class sc2reader.events.tracker.**PlayerStatsEvent** (*frames*, *data*, *build*)
Player Stats events are generated for all players that were in the game even if they've since left every 10 seconds. An additional set of stats events are generated at the end of the game.

When a player leaves the game, a single PlayerStatsEvent is generated for that player and no one else. That player continues to generate PlayerStatsEvents at 10 second intervals until the end of the game.

In 1v1 games, the above behavior can cause the losing player to have 2 events generated at the end of the game. One for leaving and one for the end of the game.

class sc2reader.events.tracker.**TrackerEvent**(*frames*)
Parent class for all tracker events.

class sc2reader.events.tracker.**UnitBornEvent**(*frames*, *data*, *build*)
Generated when a unit is created in a finished state in the game. Examples include the Marine, Zergling, and Zealot (when trained from a gateway). Units that enter the game unfinished (all buildings, warped in units) generate a *UnitInitEvent* instead.

Unfortunately, units that are born do not have events marking their beginnings like *UnitInitEvent* and *UnitDoneEvent* do. The closest thing to it are the CommandEvent game events where the command is a train unit command.

class sc2reader.events.tracker.**UnitDiedEvent**(*frames*, *data*, *build*)
Generated when a unit dies or is removed from the game for any reason. Reasons include morphing, merging, and getting killed.

class sc2reader.events.tracker.**UnitDoneEvent**(*frames*, *data*, *build*)
The counter part to the *UnitInitEvent*, generated by the game engine when an initiated unit is completed. E.g. warp-in finished, building finished, morph complete.

class sc2reader.events.tracker.**UnitInitEvent**(*frames*, *data*, *build*)
The counter part to *UnitDoneEvent*, generated by the game engine when a unit is initiated. This applies only to units which are started in game before they are finished. Primary examples being buildings and warp-in units.

class sc2reader.events.tracker.**UnitOwnerChangeEvent**(*frames*, *data*, *build*)
Generated when either ownership or control of a unit is changed. Neural Parasite is an example of an action that would generate this event.

class sc2reader.events.tracker.**UnitPositionsEvent**(*frames*, *data*, *build*)
Generated every 15 seconds. Marks the positions of the first 255 units that were damaged in the last interval. If more than 255 units were damaged, then the first 255 are reported and the remaining units are carried into the next interval.

class sc2reader.events.tracker.**UnitTypeChangeEvent**(*frames*, *data*, *build*)
Generated when the unit's type changes. This generally tracks upgrades to buildings (Hatch, Lair, Hive) and mode switches (Sieging Tanks, Phasing prisms, Burrowing roaches). There may be some other situations where a unit transformation is a type change and not a new unit.

class sc2reader.events.tracker.**UpgradeCompleteEvent**(*frames*, *data*, *build*)
Generated when a player completes an upgrade.

S

M

N

O

P

R

S

T

U

V