python-valve Documentation

Release 0.0.1

Oliver Ainsworth

December 25, 2015

Contents

1 Interacting with Source Servers **3**
- 1.1 Example . . . 4
- 1.2 Utilities . . . 5

2 Querying the Source Master Server **7**
- 2.1 valve.source.master_server . . . 7
- 2.2 Example . . . 9

3 SteamIDs **11**
- 3.1 The `SteamID` Class . . . 11
- 3.2 Exceptions . . . 13
- 3.3 Useful Constants . . . 13

4 Source Remote Console (RCON) **15**
- 4.1 Example . . . 15
- 4.2 The `RCON` Class . . . 15
- 4.3 RCON Messages . . . 16
- 4.4 REPL via `shell()` . . . 17

5 Steam Web API **19**
- 5.1 Low-level Wrapper . . . 19
- 5.2 Interfaces . . . 21

6 License **27**
- 6.1 Trademarks . . . 27

7 Indices and tables **29**

Python Module Index **31**

python-valve is a Python library which aims to provide the ability to interace with various Valve services and products, including: the Steam web API, locally installed Steam clients, Source servers and the Source master server.

Contents:

CHAPTER 1

Interacting with Source Servers

Source provides the "A2S" protocol for querying game servers. This protocol is used by the Steam and in-game server browsers to list information about servers such as their name, player count and whether or not they're password protected. `valve.source.server` provides a client implementation of A2S.

class `valve.source.a2s.`**`ServerQuerier`**(*address*, *timeout=5.0*)

Implements the A2S Source server query protocol

https://developer.valvesoftware.com/wiki/Server_queries

`get_info`()

Retreive information about the server state

This returns the response from the server which implements `__getitem__` for accessing response fields. For example:

```
server = ServerQuerier(...)
print server.get_info()["server_name"]
```

The following fields are available on the response:

Field	Description
response_type	Always `0x49`
server_name	The name of the server
map	The name of the map being ran by the server
folder	The *gamedir* if the modification being ran by the server. E.g. `tf`, `cstrike`, `csgo`.
game	A string identifying the game being ran by the server
app_id	The numeric application ID of the game ran by the server. Note that this is the app ID of the client, not the server. For example, for Team Fortress 2 `440` is returned instead of `232250` which is the ID of the server software.
player_count	Number of players currently connected
max_players	The number of player slots available. Note that `player_count` may exceed this value under certain circumstances.
bot_count	The number of AI players present
server_type	A `util.ServerType` instance representing the type of server. E.g. Dedicated, non-dedicated or Source TV relay.
platform	A :class'..util.Platform' instances represneting the platform the server is running on. E.g. Windows, Linux or Mac OS X.
password_protected	Whether or not a password is required to connect to the server.
vac_enabled	Whether or not Valve anti-cheat (VAC) is enabled
version	The version string of the server software

Currently the *extra data field* (EDF) is not supported.

get_players()
Retrive a list of all players connected to the server

The following fields are available on the response:

Field	Description
response_type	Always `0x44`
player_count	The number of players listed
players	A list of player entries

The `players` field is a list that contains `player_count` number of `messages.PlayerEntry` instances which have the same interface as the top-level response object that is returned.

The following fields are available on each player entry:

Field	Description
name	The name of the player
score	Player's score at the time of the request. What this relates to is dependant on the gamemode of the server.
duration	Number of seconds the player has been connected as a float

get_rules()
Retreive the server's game mode configuration

This method allows you capture a subset of a server's console variables (often referred to as 'cvars',) specifically those which have the `FCVAR_NOTIFY` flag set on them. These cvars are used to indicate game mode's configuration, such as the gravity setting for the map or whether friendly fire is enabled or not.

The following fields are available on the response:

Field	Description
response_type	Always `0x56`
rule_count	The number of rules
rules	A dictionary mapping rule names to their corresponding string value

ping()
Ping the server, returning the round-trip latency in milliseconds

The A2A_PING request is deprecated so this actually sends a A2S_INFO request and times that. The time difference between the two should be negligble.

1.1 Example

In this example we will query a server, printing out it's name and the number of players currently conected. Then we'll print out all the players sorted score-decesending.

```
import valve.source.a2s

SERVER_ADDRESS = (..., ...)

server = valve.source.a2s.ServerQuerier(SERVER_ADDRESS)
info = server.get_info()
players = server.get_players()

print "{player_count}/{max_players} {server_name}".format(**info)
```

```
for player in sorted(players["players"],
                     key=lambda p: p["score"], reverse=True):
    print "{score} {name}".format(**player)
```

1.2 Utilities

valve.source.util provides a handful of utility classes which are used when querying Source servers.

class valve.source.util.**Platform**(*value*)

A Source server platform identifier

This class provides utilities for representing Source server platforms as returned from a A2S_INFO request. Each platform is ultimately represented by one of the following integers:

ID	Platform
76	Linux
108	Linux
109	Mac OS X
111	Mac OS X
119	Windows

Note: Starbound uses 76 instead of 108 for Linux in the old GoldSource style.

__eq__(*other*)

Check for equality between two platforms

If `other` is not a Platform instance then an attempt is made to convert it to one using same approach as *__init__()*. This means platforms can be compared against integers and strings. For example:

```
>>>Platform(108) == "linux"
True
>>>Platform(109) == 109
True
>>>Platform(119) == "w"
True
```

Despite the fact there are two numerical identifers for Mac (109 and 111) comparing either of them together will yield `True`.

```
>>>Platform(109) == Platform(111)
True
```

__init__(*value*)

Initialise the platform identifier

The given `value` will be mapped to a numeric identifier. If the value is already an integer it must then it must exist in the table above else ValueError is returned.

If `value` is a one character long string then it's ordinal value as given by `ord()` is used. Alternately the string can be either of the following:

- Linux
- Mac OS X
- Windows

__weakref__

list of weak references to the object (if defined)

os_name
Convenience mapping to names returned by `os.name`

class `valve.source.util.`**ServerType**(*value*)
A Source server platform identifier

This class provides utilities for representing Source server types as returned from a A2S_INFO request. Each server type is ultimately represented by one of the following integers:

ID	Server type
68	Dedicated
100	Dedicated
108	Non-dedicated
112	SourceTV

Note: Starbound uses 68 instead of 100 for a dedicated server in the old GoldSource style.

__eq__(*other*)
Check for equality between two server types

If `other` is not a ServerType instance then an attempt is made to convert it to one using same approach as *__init__()*. This means server types can be compared against integers and strings. For example:

```
>>>Server(100) == "dedicated"
True
>>>Platform(108) == 108
True
>>>Platform(112) == "p"
True
```

__init__(*value*)
Initialise the server type identifier

The given `value` will be mapped to a numeric identifier. If the value is already an integer it must then it must exist in the table above else ValueError is returned.

If `value` is a one character long string then it's ordinal value as given by `ord()` is used. Alternately the string can be either of the following:

- Dedicated
- Non-Dedicated
- SourceTV

__weakref__
list of weak references to the object (if defined)

CHAPTER 2

Querying the Source Master Server

When a Source server starts it can optionally add it self to an index of live servers to enable players to find the server via matchmaking and the in-game server browsers. It does this by registering it self with the "master server". The master server is hosted by Valve but the protocol used to communicate with it is *reasonably* well documented.

Clients can request a list of server addresses from the master server for a particular region. Optionally, they can also specify a filtration criteria to restrict what servers are returned. *valve.source.master_server* provides an interface for interacting with the master server.

Note: Although "master server" is used in a singular context there are in fact multiple servers. By default *valve.source.master_server.MasterServerQuerier* will lookup `hl2master.steampowered.com` which, at the time of writing, has three `A` entries.

2.1 valve.source.master_server

class `valve.source.master_server.`**MasterServerQuerier**(*address=(u'hl2master.steampowered.com', 27011)*, *timeout=10.0*)

Implements the Source master server query protocol

https://developer.valvesoftware.com/wiki/Master_Server_Query_Protocol

__iter__()

An unfitlered iterator of all Source servers

This will issue a request for an unfiltered set of server addresses for each region. Addresses are received in batches but returning a completely unfiltered set will still take a long time and be prone to timeouts.

See *find()* for making filtered requests.

find(*region=u'all'*, ***filters*)

Find servers for a particular region and set of filtering rules

This returns an iterator which yields `(host, port)` server addresses from the master server.

`region` spcifies what regions to restrict the search to. It can either be a `REGION_` constant or a string identifying the region. Alternately a list of the strings or `REGION_` constants can be used for specifying multiple regions.

The following region identification strings are supported:

String	Region(s)
na-east	East North America
na-west	West North America
na	East North American, West North America
sa	South America
eu	Europe
as	Asia, the Middle East
oc	Oceania/Australia
af	Africa
rest	Unclassified servers
all	All of the above

Note: "`rest`" corresponds to all servers that don't fit with any other region. What causes a server to be placed in this region by the master server isn't entirely clear.

The region strings are not case sensitive. Specifying an invalid region identifier will raise a ValueError.

As well as region-based filtering, alternative filters are supported which are documented on the Valve developer wiki.

https://developer.valvesoftware.com/wiki/Master_Server_Query_Protocol#Filter

This method accepts keyword arguments which are used for building the filter string that is sent along with the request to the master server. Below is a list of all the valid keyword arguments:

Filter	Description
type	Server type, e.g. "dedicated". This can be a `ServerType` instance or any value that can be converted to a `ServerType`.
secure	Servers using Valve anti-cheat (VAC). This should be a boolean.
gamedir	A string specifying the mod being ran by the server. For example: `tf`, `cstrike`, `csgo`, etc..
map	Which map the server is running.
linux	Servers running on Linux. Boolean.
empty	Servers which are not empty. Boolean.
full	Servers which are full. Boolean.
proxy	SourceTV relays only. Boolean.
napp	Servers not running the game specified by the given application ID. E.g. `440` would exclude all TF2 servers.
noplayers	Servers that are empty. Boolean
white	Whitelisted servers only. Boolean.
gametype	Server which match *all* the tags given. This should be set to a list of strings.
gamedata	Servers which match *all* the given hidden tags. Only applicable for L4D2 servers.
gamedataor	Servers which match *any* of the given hidden tags. Only applicable to L4D2 servers.

Note: Your mileage may vary with some of these filters. There's no real guarantee that the servers returned by the master server will actually satisfy the filter. Because of this it's advisable to explicitly check for compliance by querying each server individually. See *`valve.source.a2s`*.

2.2 Example

In this example we will list all European and Asian Team Fortress 2 servers running the map *ctf_2fort* and print out their addresses.

```
import valve.source.server
import valve.source.master_server

msq = valve.source.master_server.MasterServerQuerier()
try:
    for address in msq.find(region=["eu", "as"],
                            gamedir="tf",
                            map="ctf_2fort"):
        print "{0}:{1}".format(*address)
except valve.source.server.NoResponseError:
    print "Master server request timed out!"
```

CHAPTER 3

SteamIDs

SteamID are used in many places within Valve services to identify entities such as users, groups and game servers. SteamIDs have many different representations which all need to be handled so the *valve.steam.id* module exists to provide an mechanism for representing these IDs in a usable fashion.

3.1 The SteamID Class

Rarely will you ever want to instantiate a *SteamID* directly. Instead it is best to use the *SteamID.from_community_url()* and *SteamID.from_text()* class methods for creating new instances.

class valve.steam.id.**SteamID**(*account_number*, *instance*, *type*, *universe*)

Represents a SteamID

A SteamID is broken up into four components: a 32 bit account number, a 20 bit "instance" identifier, a 4 bit account type and an 8 bit "universe" identifier.

There are 10 known accounts types as listed below. Generally you won't encounter types other than "individual" and "group".

Type	Numeric value	Can be mapped to URL	Constant
Invalid	0	No	TYPE_INVALID
Individual	1	Yes	TYPE_INDIVIDUAL
Multiseat	2	No	TYPE_MULTISEAT
Game server	3	No	TYPE_GAME_SERVER
Anonymous game server	4	No	TYPE_ANON_GAME_SERVER
Pending	5	No	TYPE_PENDING
Content server	6	No	TYPE_CONTENT_SERVER
Group	7	Yes	TYPE_CLAN
Chat	8	No	TYPE_CHAT
"P2P Super Seeder"	9	No	TYPE_P2P_SUPER_SEEDER
Anonymous user	10	No	TYPE_ANON_USER

TYPE_-prefixed constants are provided by the *valve.steam.id* module for the numerical values of each type.

All SteamIDs can be represented textually as well as by their numerical components. This is typically in the STEAM_X:Y:Z form where X, Y, Z are the "universe", "instance" and the account number respectively. There are two special cases however. If the account type if invalid then "UNKNOWN" is the textual representation. Similarly "STEAM_ID_PENDING" is used when the type is pending.

As well as the the textual representation of SteamIDs there are also the 64 and 32 bit versions which contain the SteamID components encoded into integers of corresponding width. However the 32-bit representation also includes a letter to indicate account type.

__int__()
The 64 bit representation of the SteamID

64 bit SteamIDs are only valid for those with the type *TYPE_INDIVIDUAL* or *TYPE_CLAN*. For all other types *SteamIDError* will be raised.

The 64 bit representation is calculated by multiplying the account number by two then adding the "instance" and then adding another constant which varies based on the account type.

For *TYPE_INDIVIDUAL* the constant is `0x0110000100000000`, whereas for *TYPE_CLAN* it's `0x0170000000000000`.

__str__()
The textual representation of the SteamID

This is in the STEAM_X:Y:Z form and can be parsed by *from_text()* to produce an equivalent instance. Alternately `STEAM_ID_PENDING` or `UNKNOWN` may be returned if the account type is *TYPE_PENDING* or *TYPE_INVALID* respectively.

Note: *from_text()* will still handle the `STEAM_ID_PENDING` and `UNKNOWN` cases.

__weakref__
list of weak references to the object (if defined)

as_32()
Returns the 32 bit community ID as a string

This is only applicable for *TYPE_INDIVIDUAL*, *TYPE_CLAN* and *TYPE_CHAT* types. For any other types, attempting to generate the 32-bit representation will result in a *SteamIDError* being raised.

as_64()
Returns the 64 bit representation as a string

This is only possible if the ID type is *TYPE_INDIVIDUAL* or *TYPE_CLAN*, otherwise *SteamIDError* is raised.

base_community_url = 'http://steamcommunity.com/'
Used for building community URLs

community_url(*id64=True*)
Returns the full URL to the Steam Community page for the SteamID

This can either be generate a URL from the 64 bit representation (the default) or the 32 bit one. Generating community URLs is only supported for IDs of type *TYPE_INDIVIDUAL* and *TYPE_CLAN*. Attempting to generate a URL for any other type will result in a *SteamIDError* being raised.

classmethod from_community_url(*id*, *universe=0*)
Parse a Steam community URL into a *SteamID* instance

This takes a Steam community URL for a profile or group and converts it to a SteamID. The type of the ID is infered from the type character in 32-bit community urls (`[U:1:1]` for example) or from the URL path (`/profile` or `/groups`) for 64-bit URLs.

As there is no way to determine the universe directly from URL it must be explitcly set, defaulting to *UNIVERSE_INDIVIDUAL*.

Raises *SteamIDError* if the URL cannot be parsed.

classmethod **from_text**(*id*, *type=1*)
Parse a SteamID in the STEAM_X:Y:Z form

Takes a teaxtual SteamID in the form STEAM_X:Y:Z and returns a corresponding *SteamID* instance. The X represents the account's 'universe,' Z is the account number and Y is either 1 or 0.

As the account type cannot be directly inferred from the SteamID it must be explicitly specified, defaulting to *TYPE_INDIVIDUAL*.

The two special IDs STEAM_ID_PENDING and UNKNOWN are also handled returning SteamID instances with the appropriate types set (*TYPE_PENDING* and *TYPE_INVALID* respectively) and with all other components of the ID set to zero.

type_name
The account type as a string

3.2 Exceptions

exception valve.steam.id.**SteamIDError**
Bases: exceptions.ValueError

Raised when parsing or building invalid SteamIDs

3.3 Useful Constants

As well as providing the *SteamID* class, the *valve.steam.id* module also contains numerous constants which relate to the contituent parts of a SteamID. These constants map to their numeric equivalent.

3.3.1 Account Types

The following are the various account types that can be encoded into a SteamID. Many of them are seemingly no longer in use – at least not in public facing services – and you're only likely to come across *TYPE_INDIVIDUAL*, *TYPE_CLAN* and possibly *TYPE_GAME_SERVER*.

valve.steam.id.**TYPE_INVALID = 0**

valve.steam.id.**TYPE_INDIVIDUAL = 1**

valve.steam.id.**TYPE_MULTISEAT = 2**

valve.steam.id.**TYPE_GAME_SERVER = 3**

valve.steam.id.**TYPE_ANON_GAME_SERVER = 4**

valve.steam.id.**TYPE_PENDING = 5**

valve.steam.id.**TYPE_CONTENT_SERVER = 6**

valve.steam.id.**TYPE_CLAN = 7**

valve.steam.id.**TYPE_CHAT = 8**

valve.steam.id.**TYPE_P2P_SUPER_SEEDER = 9**

valve.steam.id.**TYPE_ANON_USER = 10**

3.3.2 Universes

A SteamID "universe" provides a way of grouping IDs. Typically you'll only ever come across the *UNIVERSE_INDIVIDUAL* universe.

valve.steam.id.**UNIVERSE_INDIVIDUAL = 0**

valve.steam.id.**UNIVERSE_PUBLIC = 1**

valve.steam.id.**UNIVERSE_BETA = 2**

valve.steam.id.**UNIVERSE_INTERNAL = 3**

valve.steam.id.**UNIVERSE_DEV = 4**

valve.steam.id.**UNIVERSE_RC = 5**

CHAPTER 4

Source Remote Console (RCON)

The remote console (RCON) is available in all Source Dedicated Servers and it provides a way for server operators to access and administer their servers remotely. The *valve.source.rcon* module provides an implementation of the RCON protocol.

RCON is a request-response TCP based protocol with a simple authentication mechanism. The client initiates a connection with the server and attempts to authenticate by submitting a password. If authentication suceeds then the client is free to send further requests to the server in the same manner as you may do using the Source in-game console.

Warning: RCON does not use secure transport so the password is sent as plain text.

Note: Many RCON authentication failures in a row from a single host will result in the Source server automatically banning that IP, preventing any subsequent connection attempts.

4.1 Example

```
from valve.source.rcon import RCON

SERVER_ADDRESS = ("...", 27015)
PASSWORD = "top_secret"

with RCON(SERVER_ADDRESS, PASSWORD) as rcon:
    print(rcon("echo Hello, world!"))
```

In this example a *RCON* instance is created to connect to a Source RCON server, authenticating using the given password. Then the `echo` RCON command is issued which simply prints out what it receives.

Using the *RCON* object with the `with` statement means creation and clean up of the underlying TCP socket will happen automatically. Also, if the password is specified, the client will authenticate immediately after connecting.

4.2 The RCON Class

The *RCON* class implements the RCON client protocol. It supports the ability to finely grain transport creation, connection, authentication and clean up although its encouraged to make use of the `with` statement as shown in the example above.

class `valve.source.rcon.`**RCON**(*address*, *password=None*, *timeout=10.0*)

__call__(*command*)
Execute a command on the server

This wraps *execute()* but returns the response body instead of the request *Message* object.

__enter__()
Connect and optionally authenticate to the server

Authentication will only be attempted if the `password` attribute is set.

__exit__(*exc_type*, *exc_value*, *exc_tb*)
Disconnect from the server

__weakref__
list of weak references to the object (if defined)

authenticate(*password*)
Authenticates with the server using the given password.

Raises AuthenticationError if password is incorrect. Note that multiple attempts with the wrong password will result in the server automatically banning 'this' IP.

connect()
Connect to host, creating transport if necessary

execute(*command*, *block=True*)
Executes a SRCDS console command.

Returns the Message object that makes up the request sent to the server. If block is True, the response attribute will be set, unless a NoResposneError was raised whilst waiting for a response.

If block is False, calls must be made to process() until a response is recieved. E.g. use response_to().

Requires that the client is authenticated, otherwise an AuthenticationError is raised.

process()
Reads all avilable data from socket and attempts to process a response. Responses are automatically attached to their corresponding request.

request(*type*, *body=u''*)
Send a message to server.

If type is SEVERDATA_EXECCOMAND an addtional SERVERDATA_RESPONSE_VALUE is sent in order to facilitate correct processing of multi-packet responses.

response_to(*request*, *timeout=None*)
Returns a context manager that waits up to a given time for a response to a specific request. Assumes the request has actually been sent to an RCON server.

If the timeout period is exceeded, NoResponseError is raised.

4.3 RCON Messages

RCON *requests* and *responses* are generalised as *messages* in the python-valve implementation. If you're using *RCON.__call__()* then you won't need to worry about handling individual messages. However, *RCON.execute()* returns these raw messages so their structure is documented below.

class `valve.source.rcon.`**Message**(*id*, *type*, *body=u''*)

__weakref__
list of weak references to the object (if defined)

classmethod decode(*buffer*)
Will attempt to decode a single message from a byte buffer, returning a corresponding Message instance and the remaining buffer contents if any.

If buffer is does not contain at least one full message, IncompleteMessageError is raised.

encode()
Encode the message to a bytestring

Each packed message inludes the payload size (in bytes,) message ID and message type encoded into a 12 byte header. The header is followed by a null-terimnated ASCII-encoded string and a further trailing null terminator.

size
Packet size in bytes, minus the 'size' fields (4 bytes).

4.4 REPL via `shell()`

A small convenience function is provided by the *valve.source.rcon* module for creating command-line REPL interfaces for RCON connections.

valve.source.rcon.**shell**(*rcon=None*)

CHAPTER 5

Steam Web API

The Steam Web API provides a mechanism to use Steam services over an HTTP. The API is divided up into "interfaces" with each interface having a number of methods that can be performed on it. Python-valve provides a thin wrapper on top of these interfaces as well as a higher-level implementation.

Generally you'll want to use the higher-level interface to the API as it provides greater abstraction and session management. However the higher-level API only covers a few core interfaces of the Steam Web API, so it may be necessary to use the wrapper layer in some circumstances.

Although an API key is not strictly necessary to use the Steam Web API, it is advisable to get an API key. Using an API key allows access to greater functionality. Also, before using the Steam Web API it is good idea to read the Steam Web API Terms of Use and Steam Web API Documentation.

5.1 Low-level Wrapper

The Steam Web API is self-documenting via the `/ISteamWebAPIUtil/GetSupportedAPIList/v1/` endpoint. This enables python-valve to build the wrapper entirely automatically, which includes validating parameters and automatic generation of documentation.

The entry-point for using the API wrapper is by constructing a *API* instance. During initialisation a request is issued to the `GetSupportedAPIList` endpoint and the interfaces are constructed. If a Steam Web API key is specified then a wider selection of interfaces will be available. Note that this can be a relatively time consuming process as the response returned by `GetSupportedAPIList` can be quite large. This is especially true when an API key is given as there are more interfaces to generated.

An instance of each interface is created and bound to the *API* instance, as it is this *API* instance that will be responsible for dispatching the HTTP requests. The interfaces are made available via *API.__getitem__()*. The interface objects have methods which correspond to those returned by `GetSupportedAPIList`.

class `valve.steam.api.interface.`**API**(*key=None*, *format=u'json'*, *versions=None*, *interfaces=None*)

__getitem__(*interface_name*)
Get an interface instance by name

__init__(*key=None*, *format=u'json'*, *versions=None*, *interfaces=None*)
Initialise an API wrapper

The API is usable without an API key but exposes significantly less functionality, therefore it's advisable to use a key.

Response formatters are callables which take the Unicode response from the Steam Web API and turn it into a more usable Python object, such as dictionary. The Steam API it self can generate responses in either

JSON, XML or VDF. The formatter callables should have an attribute `format` which is a string indicating which textual format they handle. For convenience the `format` parameter also accepts the strings `json`, `xml` and `vdf` which are mapped to the *`json_format()`*, *`etree_format()`* and *`vdf_format()`* formatters respectively.

The `interfaces` argument can optionally be set to a module containing `BaseInterface` subclasses which will be instantiated and bound to the *`API`* instance. If not given then the interfaces are loaded using `ISteamWebAPIUtil/GetSupportedAPIList`.

The optional `versions` argument allows specific versions of interface methods to be used. If given, `versions` should be a mapping of further mappings keyed against the interface name. The inner mapping should specify the version of interface method to use which is keyed against the method name. These mappings don't need to be complete and can omit methods or even entire interfaces. In which case the default behaviour is to use the method with the highest version number.

Parameters

- **`key`** (*str*) – a Steam Web API key.
- **`format`** – response formatter.
- **`versions`** – the interface method versions to use.
- **`interfaces`** – a module containing `BaseInterface` subclasses or `None` if they should be loaded for the first time.

`api_root` = u'https://api.steampowered.com/'

`request`(*http_method*, *interface*, *method*, *version*, *params=None*, *format=None*)
Issue a HTTP request to the Steam Web API

This is called indirectly by interface methods and should rarely be called directly. The response to the request is passed through the response formatter which is then returned.

Parameters

- **`interface`** (*str*) – the name of the interface.
- **`method`** (*str*) – the name of the method on the interface.
- **`version`** (*int*) – the version of the method.
- **`params`** – a mapping of GET or POST data to be sent with the request.
- **`format`** – a response formatter callable to overide `format`.

`session`(**args*, ***kwds*)
Create an API sub-session without rebuilding the interfaces

This returns a context manager which yields a new *`API`* instance with the same interfaces as the current one. The difference between this and creating a new *`API`* manually is that this will avoid rebuilding the all interface classes which can be slow.

`versions`()
Get the versions of the methods for each interface

This returns a dictionary of dictionaries which is keyed against interface names. The inner dictionaries map method names to method version numbers. This structure is suitable for passing in as the `versions` argument to *`__init__()`*.

5.1.1 Interface Method Version Pinning

It's important to be aware of the fact that API interface methods can have multiple versions. For example, `ISteamApps/GetAppList`. This means they may take different arguments and returned different responses. The default behaviour of the API wrapper is to always expose the method with the highest version number.

This is fine in most cases, however it does pose a potential problem. New versions of interface methods are likely to break backwards compatability. Therefore *API* provides a mechanism to manually specify the interface method versions to use via the `versions` argument to *API.__init__()*.

The if given at all, `versions` is expected to be a dictionary of dictionaries keyed against interface names. The inner dictionaries map method names to versions. For example:

```
{"ISteamApps": {"GetAppList": 1}}
```

Passsing this into *API.__init__()* would mean version 1 of `ISteamApps/GetAppList` would be used in preference to the default behaviour of using the highest version – wich at the time of writing is version 2.

It is important to pin your interface method versions when your code enters production or otherwise face the risk of it breaking in the future if and when Valve updates the Steam Web API. The `API.pin_versions()` method is provided to help in determining what versions to pin. How to integrate interface method version pinning into existing code is an excerise for the reader however.

5.1.2 Response Formatters

`valve.steam.api.interface.`**`json_format`**(*response*)
Parse response as JSON using the standard Python JSON parser

Returns the JSON object encoded in the response.

`valve.steam.api.interface.`**`etree_format`**(*response*)
Parse response using ElementTree

Returns a `xml.etree.ElementTree.Element` of the root element of the response.

`valve.steam.api.interface.`**`vdf_format`**(*response*)
Parse response using `valve.vdf`

Returns a dictionary decoded from the VDF.

5.2 Interfaces

These interfaces are automatically wrapped and documented. The availability of some interfaces is dependant on whether or not an API key is given. It should also be noted that as the interfaces are generated automatically they do not respect the naming conventions as detailed in PEP 8.

class `interfaces.`**`IGCVersion_205790`**(*api*)

`GetClientVersion`()

`GetServerVersion`()

`name` **= u'IGCVersion_205790'**

class `interfaces.`**`IGCVersion_440`**(*api*)

`GetClientVersion`()

GetServerVersion()

name = u'IGCVersion_440'

class interfaces.**IGCVersion_570**(*api*)

GetClientVersion()

GetServerVersion()

name = u'IGCVersion_570'

class interfaces.**IGCVersion_730**(*api*)

GetServerVersion()

name = u'IGCVersion_730'

class interfaces.**IPortal2Leaderboards_620**(*api*)

GetBucketizedData(*leaderboardName*)

Parameters **leaderboardName** (*string*) – The leaderboard name to fetch data for.

name = u'IPortal2Leaderboards_620'

class interfaces.**IPortal2Leaderboards_841**(*api*)

GetBucketizedData(*leaderboardName*)

Parameters **leaderboardName** (*string*) – The leaderboard name to fetch data for.

name = u'IPortal2Leaderboards_841'

class interfaces.**ISteamApps**(*api*)

GetAppList()

GetServersAtAddress(*addr*)

Parameters **addr** (*string*) – IP or IP:queryport to list

UpToDateCheck(*appid*, *version*)

Parameters

- **appid** (*uint32*) – AppID of game
- **version** (*uint32*) – The installed version of the game

name = u'ISteamApps'

class interfaces.**ISteamDirectory**(*api*)

GetCMList(*cellid*, *maxcount=None*)

Parameters

- **cellid** (*uint32*) – Client's Steam cell ID
- **maxcount** (*uint32*) – Max number of servers to return

name = u'ISteamDirectory'

class interfaces.**ISteamEnvoy**(*api*)

PaymentOutNotification()

PaymentOutReversalNotification()

name = u'ISteamEnvoy'

class interfaces.**ISteamNews**(*api*)

GetNewsForApp(*appid*, *count=None*, *enddate=None*, *feeds=None*, *maxlength=None*)

Parameters

- **appid** (*uint32*) – AppID to retrieve news for
- **count** (*uint32*) – # of posts to retrieve (default 20)
- **enddate** (*uint32*) – Retrieve posts earlier than this date (unix epoch timestamp)
- **feeds** (*string*) – Comma-seperated list of feed names to return news for
- **maxlength** (*uint32*) – Maximum length for the content to return, if this is 0 the full content is returned, if it's less then a blurb is generated to fit.

name = u'ISteamNews'

class interfaces.**ISteamPayPalPaymentsHub**(*api*)

PayPalPaymentsHubPaymentNotification()

name = u'ISteamPayPalPaymentsHub'

class interfaces.**ISteamRemoteStorage**(*api*)

GetCollectionDetails(*collectioncount*, *publishedfileids0*)

Parameters

- **collectioncount** (*uint32*) – Number of collections being requested
- **publishedfileids0** (*uint64*) – collection ids to get the details for

GetPublishedFileDetails(*itemcount*, *publishedfileids0*)

Parameters

- **itemcount** (*uint32*) – Number of items being requested
- **publishedfileids0** (*uint64*) – published file id to look up

name = u'ISteamRemoteStorage'

class interfaces.**ISteamUserAuth**(*api*)

AuthenticateUser(*encrypted_loginkey*, *sessionkey*, *steamid*)

Parameters

- **encrypted_loginkey** (*rawbinary*) – Should be the users hashed loginkey, AES encrypted with the sessionkey.

- **sessionkey** (*rawbinary*) – Should be a 32 byte random blob of data, which is then encrypted with RSA using the Steam system's public key. Randomness is important here for security.
- **steamid** (*uint64*) – Should be the users steamid, unencrypted.

name = u'ISteamUserAuth'

class interfaces.**ISteamUserOAuth**(*api*)

GetTokenDetails(*access_token*)

Parameters **access_token** (*string*) – OAuth2 token for which to return details

name = u'ISteamUserOAuth'

class interfaces.**ISteamUserStats**(*api*)

GetGlobalAchievementPercentagesForApp(*gameid*)

Parameters **gameid** (*uint64*) – GameID to retrieve the achievement percentages for

GetGlobalStatsForGame(*appid*, *count*, *name0*, *enddate=None*, *startdate=None*)

Parameters

- **appid** (*uint32*) – AppID that we're getting global stats for
- **count** (*uint32*) – Number of stats get data for
- **enddate** (*uint32*) – End date for daily totals (unix epoch timestamp)
- **name0** (*string*) – Names of stat to get data for
- **startdate** (*uint32*) – Start date for daily totals (unix epoch timestamp)

GetNumberOfCurrentPlayers(*appid*)

Parameters **appid** (*uint32*) – AppID that we're getting user count for

name = u'ISteamUserStats'

class interfaces.**ISteamWebAPIUtil**(*api*)

GetServerInfo()

GetSupportedAPIList()

name = u'ISteamWebAPIUtil'

class interfaces.**ISteamWebUserPresenceOAuth**(*api*)

PollStatus(*message*, *steamid*, *umqid*, *pollid=None*, *secidletime=None*, *sectimeout=None*, *use_accountids=None*)

Parameters

- **message** (*uint32*) – Message that was last known to the user
- **pollid** (*uint32*) – Caller-specific poll id
- **secidletime** (*uint32*) – How many seconds is client considering itself idle, e.g. screen is off
- **sectimeout** (*uint32*) – Long-poll timeout in seconds

- **steamid** (*string*) – Steam ID of the user
- **umqid** (*uint64*) – UMQ Session ID
- **use_accountids** (*uint32*) – Boolean, 0 (default): return steamid_from in output, 1: return accountid_from

name = u'ISteamWebUserPresenceOAuth'

class interfaces.**IPlayerService**(*api*)

RecordOfflinePlaytime(*play_sessions*, *steamid*, *ticket*)

Parameters

- **play_sessions** (*string*) –
- **steamid** (*uint64*) –
- **ticket** (*string*) –

name = u'IPlayerService'

class interfaces.**IAccountRecoveryService**(*api*)

ReportAccountRecoveryData(*install_config*, *loginuser_list*, *machineid*, *shasentryfile*)

Parameters

- **install_config** (*string*) –
- **loginuser_list** (*string*) –
- **machineid** (*string*) –
- **shasentryfile** (*string*) –

RetrieveAccountRecoveryData(*requesthandle*)

Parameters **requesthandle** (*string*) –

name = u'IAccountRecoveryService'

Although Python libraries *do* already exist for many aspects which python-valve aims to cover, many of them are aging and no long maintained. python-valve hopes to change that and provide an all-in-one library for interfacing with Valve products and services that is well tested, well documented and actively maintained.

python-valve's functional test suite for its A2S implentation is actively ran against thousands of servers to ensure that if any subtle changes are made by Valve that break things they can be quickly picked up and fixed.

CHAPTER 6

License

6.1 Trademarks

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

i

interfaces, 21

v

valve.source.a2s, 3
valve.source.master_server, 7
valve.source.rcon, 14
valve.source.util, 5
valve.steam.api.interface, 19
valve.steam.id, 9

Symbols

__call__() (valve.source.rcon.RCON method), 15
__enter__() (valve.source.rcon.RCON method), 16
__eq__() (valve.source.util.Platform method), 5
__eq__() (valve.source.util.ServerType method), 6
__exit__() (valve.source.rcon.RCON method), 16
__getitem__() (valve.steam.api.interface.API method), 19
__init__() (valve.source.util.Platform method), 5
__init__() (valve.source.util.ServerType method), 6
__init__() (valve.steam.api.interface.API method), 19
__int__() (valve.steam.id.SteamID method), 12
__iter__() (valve.source.master_server.MasterServerQuerier method), 7
__str__() (valve.steam.id.SteamID method), 12
__weakref__ (valve.source.rcon.Message attribute), 16
__weakref__ (valve.source.rcon.RCON attribute), 16
__weakref__ (valve.source.util.Platform attribute), 5
__weakref__ (valve.source.util.ServerType attribute), 6
__weakref__ (valve.steam.id.SteamID attribute), 12

A

API (class in valve.steam.api.interface), 19
api_root (valve.steam.api.interface.API attribute), 20
as_32() (valve.steam.id.SteamID method), 12
as_64() (valve.steam.id.SteamID method), 12
authenticate() (valve.source.rcon.RCON method), 16
AuthenticateUser() (interfaces.ISteamUserAuth method), 23

B

base_community_url (valve.steam.id.SteamID attribute), 12

C

community_url() (valve.steam.id.SteamID method), 12
connect() (valve.source.rcon.RCON method), 16

D

decode() (valve.source.rcon.Message class method), 16

E

encode() (valve.source.rcon.Message method), 17
etree_format() (in module valve.steam.api.interface), 21
execute() (valve.source.rcon.RCON method), 16

F

find() (valve.source.master_server.MasterServerQuerier method), 7
from_community_url() (valve.steam.id.SteamID class method), 12
from_text() (valve.steam.id.SteamID class method), 12

G

get_info() (valve.source.a2s.ServerQuerier method), 3
get_players() (valve.source.a2s.ServerQuerier method), 4
get_rules() (valve.source.a2s.ServerQuerier method), 4
GetAppList() (interfaces.ISteamApps method), 22
GetBucketizedData() (interfaces.IPortal2Leaderboards_620 method), 22
GetBucketizedData() (interfaces.IPortal2Leaderboards_841 method), 22
GetClientVersion() (interfaces.IGCVersion_205790 method), 21
GetClientVersion() (interfaces.IGCVersion_440 method), 21
GetClientVersion() (interfaces.IGCVersion_570 method), 22
GetCMList() (interfaces.ISteamDirectory method), 22
GetCollectionDetails() (interfaces.ISteamRemoteStorage method), 23
GetGlobalAchievementPercentagesForApp() (interfaces.ISteamUserStats method), 24
GetGlobalStatsForGame() (interfaces.ISteamUserStats method), 24
GetNewsForApp() (interfaces.ISteamNews method), 23
GetNumberOfCurrentPlayers() (interfaces.ISteamUserStats method), 24

GetPublishedFileDetails() (interfaces.ISteamRemoteStorage method), 23
GetServerInfo() (interfaces.ISteamWebAPIUtil method), 24
GetServersAtAddress() (interfaces.ISteamApps method), 22
GetServerVersion() (interfaces.IGCVersion_205790 method), 21
GetServerVersion() (interfaces.IGCVersion_440 method), 21
GetServerVersion() (interfaces.IGCVersion_570 method), 22
GetServerVersion() (interfaces.IGCVersion_730 method), 22
GetSupportedAPIList() (interfaces.ISteamWebAPIUtil method), 24
GetTokenDetails() (interfaces.ISteamUserOAuth method), 24

I

IAccountRecoveryService (class in interfaces), 25
IGCVersion_205790 (class in interfaces), 21
IGCVersion_440 (class in interfaces), 21
IGCVersion_570 (class in interfaces), 22
IGCVersion_730 (class in interfaces), 22
interfaces (module), 21
IPlayerService (class in interfaces), 25
IPortal2Leaderboards_620 (class in interfaces), 22
IPortal2Leaderboards_841 (class in interfaces), 22
ISteamApps (class in interfaces), 22
ISteamDirectory (class in interfaces), 22
ISteamEnvoy (class in interfaces), 22
ISteamNews (class in interfaces), 23
ISteamPayPalPaymentsHub (class in interfaces), 23
ISteamRemoteStorage (class in interfaces), 23
ISteamUserAuth (class in interfaces), 23
ISteamUserOAuth (class in interfaces), 24
ISteamUserStats (class in interfaces), 24
ISteamWebAPIUtil (class in interfaces), 24
ISteamWebUserPresenceOAuth (class in interfaces), 24

J

json_format() (in module valve.steam.api.interface), 21

M

MasterServerQuerier (class in valve.source.master_server), 7
Message (class in valve.source.rcon), 16

N

name (interfaces.IAccountRecoveryService attribute), 25
name (interfaces.IGCVersion_205790 attribute), 21
name (interfaces.IGCVersion_440 attribute), 22
name (interfaces.IGCVersion_570 attribute), 22
name (interfaces.IGCVersion_730 attribute), 22
name (interfaces.IPlayerService attribute), 25
name (interfaces.IPortal2Leaderboards_620 attribute), 22
name (interfaces.IPortal2Leaderboards_841 attribute), 22
name (interfaces.ISteamApps attribute), 22
name (interfaces.ISteamDirectory attribute), 22
name (interfaces.ISteamEnvoy attribute), 23
name (interfaces.ISteamNews attribute), 23
name (interfaces.ISteamPayPalPaymentsHub attribute), 23
name (interfaces.ISteamRemoteStorage attribute), 23
name (interfaces.ISteamUserAuth attribute), 24
name (interfaces.ISteamUserOAuth attribute), 24
name (interfaces.ISteamUserStats attribute), 24
name (interfaces.ISteamWebAPIUtil attribute), 24
name (interfaces.ISteamWebUserPresenceOAuth attribute), 25

O

os_name (valve.source.util.Platform attribute), 5

P

PaymentOutNotification() (interfaces.ISteamEnvoy method), 23
PaymentOutReversalNotification() (interfaces.ISteamEnvoy method), 23
PayPalPaymentsHubPaymentNotification() (interfaces.ISteamPayPalPaymentsHub method), 23
ping() (valve.source.a2s.ServerQuerier method), 4
Platform (class in valve.source.util), 5
PollStatus() (interfaces.ISteamWebUserPresenceOAuth method), 24
process() (valve.source.rcon.RCON method), 16

R

RCON (class in valve.source.rcon), 15
RecordOfflinePlaytime() (interfaces.IPlayerService method), 25
ReportAccountRecoveryData() (interfaces.IAccountRecoveryService method), 25
request() (valve.source.rcon.RCON method), 16
request() (valve.steam.api.interface.API method), 20
response_to() (valve.source.rcon.RCON method), 16
RetrieveAccountRecoveryData() (interfaces.IAccountRecoveryService method), 25

S

ServerQuerier (class in valve.source.a2s), 3
ServerType (class in valve.source.util), 6
session() (valve.steam.api.interface.API method), 20

shell() (in module valve.source.rcon), 17
size (valve.source.rcon.Message attribute), 17
SteamID (class in valve.steam.id), 11
SteamIDError, 13

T

TYPE_ANON_GAME_SERVER (in module valve.steam.id), 13
TYPE_ANON_USER (in module valve.steam.id), 13
TYPE_CHAT (in module valve.steam.id), 13
TYPE_CLAN (in module valve.steam.id), 13
TYPE_CONTENT_SERVER (in module valve.steam.id), 13
TYPE_GAME_SERVER (in module valve.steam.id), 13
TYPE_INDIVIDUAL (in module valve.steam.id), 13
TYPE_INVALID (in module valve.steam.id), 13
TYPE_MULTISEAT (in module valve.steam.id), 13
type_name (valve.steam.id.SteamID attribute), 13
TYPE_P2P_SUPER_SEEDER (in module valve.steam.id), 13
TYPE_PENDING (in module valve.steam.id), 13

U

UNIVERSE_BETA (in module valve.steam.id), 14
UNIVERSE_DEV (in module valve.steam.id), 14
UNIVERSE_INDIVIDUAL (in module valve.steam.id), 14
UNIVERSE_INTERNAL (in module valve.steam.id), 14
UNIVERSE_PUBLIC (in module valve.steam.id), 14
UNIVERSE_RC (in module valve.steam.id), 14
UpToDateCheck() (interfaces.ISteamApps method), 22

V

valve.source.a2s (module), 3
valve.source.master_server (module), 7
valve.source.rcon (module), 14
valve.source.util (module), 5
valve.steam.api.interface (module), 19
valve.steam.id (module), 9
vdf_format() (in module valve.steam.api.interface), 21
versions() (valve.steam.api.interface.API method), 20