python-valve Documentation

Release 0.2.0

Oliver Ainsworth

Sep 11, 2017

Contents

python-valve is a Python library which aims to provide the ability to interface 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.a2s` 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

Note: Instantiating this class creates a socket. Be sure to close the querier once finished with it. See `valve.source.BaseQuerier`.

`info`()

Retreive information about the server state

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

```
with ServerQuerier(...) as server:
    print(server.info()["server_name"])
```

The following fields are available on the response:

Field	Description
re-sponse_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. See *`players()`* for caveats about the accuracy of this field.
max_players	The number of player slots available. Note that `player_count` may exceed this value under certain circumstances. See *`players()`*.
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 *`util.Platform`* instances represneting the platform the server is running on. E.g. Windows, Linux or Mac OS X.
pass-word_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.

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.

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.
dura-tion	Number of seconds the player has been connected as a float

Note: Under certain circumstances, some servers may return a player list which contains empty `name` fields. This can lead to `player_count` being misleading.

Filtering out players with empty names may yield a more accurate enumeration of players:

```
with ServerQuerier(...) as query:
    players = []
    for player in query.players()["players"]:
        if player["name"]:
            players.append(player)
    player_count = len(players)
```

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

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 = (..., ...)

with valve.source.a2s.ServerQuerier(SERVER_ADDRESS) as server:
    info = server.info()
    players = server.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))
```

Queriers and Exceptions

Both *`valve.source.a2s.ServerQuerier`* and *`valve.source.master_server.MasterServerQuerier`* are based on a common querier interface. They also raise similar exceptions. All of these live in the *`valve.source`* module.

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

Base class for implementing source server queriers.

When an instance of this class is initialised a socket is created. It's important that, once a querier is to be discarded, the associated socket be closed via *`close()`*. For example:

```
querier = valve.source.BaseQuerier(('...', 27015))
try:
    querier.request(...)
finally:
    querier.close()
```

When server queriers are used as context managers, the socket will be cleaned up automatically. Hence it's preferably to use the *with* statement over the *try-finally* pattern described above:

```
with valve.source.BaseQuerier(('...', 27015)) as querier:
    querier.request(...)
```

Once a querier has been closed, any attempts to make additional requests will result in a *QuerierClosedError* to be raised.

Variables

- **host** – Host requests will be sent to.
- **port** – Port number requests will be sent to.
- **timeout** – How long to wait for a response to a request.

close()

Close the querier's socket.

It is safe to call this multiple times.

get_response()

Wait for a response to a request.

Raises

- ***NoResponseError*** – If the configured `timeout` is reached before a response is received.
- ***QuerierClosedError*** – If the querier has been closed.

Returns The raw response as a `bytes`.

request(**request*)

Issue a request.

The given request segments will be encoded and combined to form the final message that is sent to the configured address.

Parameters **request** (*valve.source.messages.Message*) – Request message segments.

Raises ***QuerierClosedError*** – If the querier has been closed.

exception valve.source.**NoResponseError**

Raised when a server querier doesn't receive a response.

exception valve.source.**QuerierClosedError**

Raised when attempting to use a querier after it's closed.

Identifying Server Platforms

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.

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

Implements the Source master server query protocol

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

Note: Instantiating this class creates a socket. Be sure to close the querier once finished with it. See *`valve.source.BaseQuerier`*.

`__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.

Note: If a request times out then the iterator will terminate early. Previous versions would propagate a `NoResponseError`.

See *`find()`* for making filtered requests.

find(*region='all'*, *duplicates=<Duplicates.SKIP: 'skip'>*, ***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.
noplay-ers	Servers that are empty. Boolean
white	Whitelisted servers only. Boolean.
game-type	Server which match *all* the tags given. This should be set to a list of strings.
game-data	Servers which match *all* the given hidden tags. Only applicable for L4D2 servers.
game-dataor	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`*.

The master server may return duplicate addresses. By default, these duplicates are excldued from the iterator returned by this method. See `Duplicates` for controller this behaviour.

class `valve.source.master_server.`**`Duplicates`**

Bases: `enum.Enum`

Behaviour for duplicate addresses.

These values are intended to be used with `MasterServerQuerier.find()` to control how duplicate addresses returned by the master server are treated.

Variables

- **`KEEP`** – All addresses are returned, even duplicates.
- **`SKIP`** – Skip duplicate addresses.
- **`STOP`** – Stop returning addresses when a duplicate is encountered.

Example

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

```
import valve.source.master_server

with valve.source.master_server.MasterServerQuerier() as msq:
    servers = msq.find(
        region=["eu", "as"],
        duplicates="skip",
        gamedir="tf",
        map="ctf_2fort",
    )
    for host, port in servers:
        print "{0}:{1}".format(host, port)
```

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.

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.

`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 expliticly 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

Exceptions

exception `valve.steam.id.`**`SteamIDError`**

Bases: `ValueError`

Raised when parsing or building invalid SteamIDs

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.

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`**

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

`valve.steam.id.`**`TYPE_CHAT`** **= 8**
int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

`valve.steam.id.`**`TYPE_P2P_SUPER_SEEDER`** **= 9**
int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

`valve.steam.id.`**`TYPE_ANON_USER`** **= 10**
int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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**
int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

`valve.steam.id.`**`UNIVERSE_PUBLIC`** **= 1**
int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

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

int(x=0) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

CHAPTER 4

Source Remote Console (RCON)

Source remote console (or RCON) provides a way for server operators to administer and interact with their servers remotely in the same manner as the console provided by **srcds**. The *valve.rcon* module provides an implementation of the RCON protocol.

RCON is a simple, TCP-based request-response protocol with support for basic authentication. The RCON client initiates a connection to a server and attempts to authenticate by submitting a password. If authentication succeeds then the client is free to send further requests. These subsequent requests are interpreted the same way as if you were to type them into the **srcds** console.

Warning: Passwords and console commands are sent in plain text. Tunneling the connection through a secure channel may be advisable where possible.

Note: Multiple 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.

High-level API

The *valve.rcon* module provides a number of ways to interact with RCON servers. The simplest is the *execute()* function which executes a single command on the server and returns the response as a string.

In many cases this may be sufficient but it's important to consider that *execute()* will create a new, temporary connection for every command. If order to reuse a connection the *RCON* class should be used directly.

Also note that *execute()* only returns Unicode strings which may prove problematic in some cases. See *Unicode and String Encoding*.

valve.rcon.**execute**(*address*, *password*, *command*)

Execute a command on an RCON server.

This is a *very* high-level interface which connects to the given RCON server using the provided credentials and executes a command.

Parameters

- **`address`** – the address of the server to connect to as a tuple containing the host as a string and the port as an integer.
- **`password`** (*`str`*) – the password to use to authenticate the connection.
- **`command`** (*`str`*) – the command to execute on the server.

Raises

- **`UnicodeDecodeError`** – if the response could not be decoded into Unicode.
- **`RCONCommunicationError`** – if a connection to the RCON server could not be made.
- **`RCONAuthenticationError`** – if authentication failed, either due to being banned or providing the wrong password.
- **`RCONMessageError`** – if the response body couldn't be decoded into a Unicode string.

Returns the response to the command as a Unicode string.

Core API

The core API for the RCON implementation is split encapsulated by two distinct classes: *`RCONMessage`* and *`RCON`*.

Representing RCON Messages

Each RCON message, whether a request or a response, is represented by an instance of the *`RCONMessage`* class. Each message has three fields: the message ID, type and contents or body. The message ID of a request is reflected back to the client when the server returns a response but is otherwise unsued by this implementation. The type is one of four constants (represented by three distinct values) which signifies the semantics of the message's ID and body. The body it self is an opaque string; its value depends on the type of message.

class `valve.rcon.`**`RCONMessage`**(*id_*, *type_*, *body_or_text*)

Represents a RCON request or response.

classmethod **`decode`**(*buffer_*)

Decode a message from a bytestring.

This will attempt to decode a single message from the start of the given buffer. If the buffer contains more than a single message then this must be called multiple times.

Raises **`MessageError`** – if the buffer doesn't contain a valid message.

Returns a tuple containing the decoded *`RCONMessage`* and the remnants of the buffer. If the buffer contained exactly one message then the remaning buffer will be empty.

`encode`()

Encode message to a bytestring.

`text`

Get the body of the message as Unicode.

Raises **`UnicodeDecodeError`** – if the body cannot be decoded as ASCII.

Returns the body of the message as a Unicode string.

Note: It has been reported that some servers may not return valid ASCII as they're documented to do so. Therefore you should always handle the potential `UnicodeDecodeError`.

If the correct encoding is known you can manually decode `body` for your self.

Unicode and String Encoding

The type of the body field of RCON messages is documented as being a double null-terminated, ASCII-encoded string. At the Python level though both Unicode strings and raw byte string interfaces are provided by *`RCONMessage.text`* and `RCONMessage.body` respectively.

In Python you are encouraged to deal with text (a.k.a. Unicode strings) in preference to raw byte strings unless strictly neccessary. However, it has been reported that under some conditions RCON servers may return invalid ASCII sequences in the response body. Therefore it is possible that the textual representation of the body cannot be determined and attempts to access *`RCONMessage.text`* will fail with a `UnicodeDecodeError` being raised.

It appears – but is not conclusively determined – that RCON servers in fact return UTF-8-encoded message bodies, hence why ASCII seems to to work in most cases. Until this can be categorically proven as the behaviour that should be expected Python-valve will continue to attempt to process ASCII strings.

If you come across `UnicodeDecodeError` whilst accessing response bodies you will instead have to make-do and handle the raw byte strings manually. For example:

```
response = rcon.execute("command")
response_text = response.body.decode("utf-8")
```

If this is undesirable it is also possible to globally set the encoding used by *`RCONMessage`* but this *not* particularly encouraged:

```
import valve.rcon

valve.rcon.RCONMessage.ENCODING = "utf-8"
```

Creating RCON Connections

class `valve.rcon.`**`RCON`**(*address*, *password*, *timeout=None*)
Represents an RCON connection.

`__call__`(*command*)
Invoke a command.

This is a higher-level version of *`execute()`* that always blocks and only returns the response body.

Raises **`RCONMessageError`** – if the response body couldn't be decoded into a Unicode string.

Returns the response to the command as a Unicode string.

`authenticate`(*timeout=None*)
Authenticate with the server.

This sends an authentication message to the connected server containing the password. If the password is correct the server sends back an acknowledgement and will allow all subsequent commands to be executed.

However, if the password is wrong the server will either notify the client or immediately drop the connection depending on whether the client IP has been banned or not. In either case, the client connection will be closed and an exception raised.

Note: Client banning IP banning happens automatically after a few failed attempts at authentication. Assuming you can direct access to the server's console you can unban the client IP using the `removeip` command:

```
Banning xxx.xxx.xxx.xx for rcon hacking attempts
] removeip xxx.xxx.xxx.xxx
removeip:  filter removed for xxx.xxx.xxx.xxx
```

---

param timeout the number of seconds to wait for a response. If not given the connection-global timeout is used.

raises RCONAuthenticationError if authentication failed, either due to being banned or providing the wrong password.

raises RCONTimeoutError if the server takes too long to respond. The connection will be closed in this case as well.

Raises

- **`RCONError`** – if closed.
- **`RCONError`** – if not connected.

`authenticated`
Determine if the connection is authenticated.

`close``()`
Close connection to a server.

`closed`
Determine if the connection has been closed.

`connect``()`
Create a connection to a server.

Raises

- **`RCONError`** – if closed.
- **`RCONError`** – if connected.

`connected`
Determine if a connection has been made.

Note: Strictly speaking this does not guarantee that any subsequent attempt to execute a command will succeed as the underlying socket may be closed by the server at any time. It merely indicates that a previous call to *`connect()`* was successful.

`cvarlist``()`
Get all ConVars for an RCON connection.

This will issue a `cvarlist` command to it in order to enumerate all available ConVars.

Returns an iterator of :class:‘ConVar‘s which may be empty.

execute(*command*, *block=True*, *timeout=None*)
Invoke a command.

Invokes the given command on the conncted server. By default this will block (up to the timeout) for a response. This can be disabled if you don't care about the response.

param str command the command to execute.

param bool block whether or not to wait for a response.

param timeout the number of seconds to wait for a response. If not given the connection-global timeout is used.

raises RCONCommunicationError if the socket is closed or in any other erroneous state whilst issuing the request or receiving the response.

raises RCONTimeoutError if the timeout is reached waiting for a response. This doesn't close the connection but the response is lost.

returns the response to the command as a *RCONMessage* or `None` depending on whether `block` was `True` or not.

Raises

- **RCONError** – if not authenticated.
- **RCONError** – if not connected.

Example

```
import valve.rcon

address = ("rcon.example.com", 27015)
password = "top-secrect-password"
with valve.rcon.RCON(address, password) as rcon:
    response = rcon.execute("echo Hello, world!")
    print(response.text)
```

Command-line Client

As well as providing means to programatically interact with RCON servers, the *valve.rcon* module also provides an interactive, command-line client. A client shell can be started by calling *shell()* or running the *valve.rcon* module.

`valve.rcon.`**shell**(*address=None*, *password=None*)
A simple interactive RCON shell.

This will connect to the server identified by the given address using the given password. If a password is not given then the shell will prompt for it. If no address is given, then no connection will be made automatically and the user will have to do it manually using `!connect`.

Once connected the shell simply dispatches commands and prints the response to stdout.

Parameters

- **address** – a network address tuple containing the host and port of the RCON server.
- **password** (*str*) – the password for the server. This is ignored if `address` is not given.

Using the RCON Shell

When `shell()` is executed, an interactive RCON shell is created. This shell reads commands from stdin, passes them to a connected RCON server then prints the response to stdout in a conventional read-eval-print pattern.

By default, commands are treated as plain RCON commmands and are passed directly to the connected server for evaluation. However, commands prefixed with an exclamation mark are interpreted by the shell it self:

`!connect` Connect to an RCON server. This command accepts two space-separated arguments: the address of the server and the corresponding password; the latter is optional. If the password is not given the user is prompted for it.

If the shell is already connected to a server then it will disconnect first before connecting to the new one.

`!disconnect` Disconnect from the current RCON server.

`!shutdown` Shutdown the RCON server. This actually just sends an `exit` command to the server. This must be used instead of `exit` as its behaviour could prove confusing with `!exit` otherwise.

`!exit` Exit the shell. This *does not* shutdown the RCON server.

Help is available via the `help` command. When connected, an optional argument can be provided which is the RCON command to show help for.

When connected to a server, command completions are provided via the tab key.

Command-line Invocation

The `valve.rcon` module is runnable. When ran with no arguments its the same as calling `shell()` with defaults. As with `shell()`, the address and password can be provided as a part of the invoking command:

```
$ python -m valve.rcon
$ python -m valve.rcon rcon.example.com:27015
$ python -m valve.rcon rcon.example.com:27015 --password TOP-SECRET
```

Warning: Passing sensitive information via command-line arguments, such as your RCON password, can be *dangerous*. For example, it can show up in **`ps`** output.

Executing a Single Command

When ran, the module has two modes of execution: the default, which will spawn an interactive RCON shell and the single command execution mode. When passed the `--execute` argument, **`python -m valve.rcon`** will run the given command and exit with a status code of zero upon completion. The command response is printed to stdout.

This can be useful for simple scripting of RCON commands outside of a Python environment, such as in a shell script.

```
$ python -m valve.rcon rcon.example.com:27015 \
    --password TOP-SECRET --execute "echo Hello, world!"
```

Usage

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.

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='json'*, *versions=None*, *interfaces=None*)

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

`__init__`(*key=None*, *format='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` = **'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`()
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__()`.

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.

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.

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 = 'IGCVersion_205790'

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

`GetClientVersion`()

`GetServerVersion`()

name = 'IGCVersion_440'

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

`GetClientVersion`()

`GetServerVersion`()

name = 'IGCVersion_570'

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

`GetServerVersion`()

name = 'IGCVersion_730'

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

`GetBucketizedData`(*leaderboardName*)

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

name = 'IPortal2Leaderboards_620'

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

`GetBucketizedData`(*leaderboardName*)

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

name = '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 = '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` = **'ISteamDirectory'**

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

`PaymentOutReversalNotification`()

`name` = **'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` = **'ISteamNews'**

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` = **'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 = 'ISteamUserAuth'

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

GetTokenDetails(*access_token*)

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

name = '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 = 'ISteamUserStats'

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

GetServerInfo()

GetSupportedAPIList()

name = '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 = 'ISteamWebUserPresenceOAuth'

class interfaces.**ITFSystem_440**(*api*)

GetWorldStatus()

name = 'ITFSystem_440'

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

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

Parameters

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

name = '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 = 'IAccountRecoveryService'

Although Python libraries *do* already exist for many aspects which python-valve aims to cover, many of them are ageing 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 implementation 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

Trademarks

Valve, the Valve logo, Half-Life, the Half-Life logo, the Lambda logo, Steam, the Steam logo, Team Fortress, the Team Fortress logo, Opposing Force, Day of Defeat, the Day of Defeat logo, Counter-Strike, the Counter-Strike logo, Source, the Source logo, Counter-Strike: Condition Zero, Portal, the Portal logo, Dota, the Dota 2 logo, and Defense of the Ancients are trademarks and/or registered trademarks of Valve Corporation.

Any reference to these are purely for the purpose of identification. Valve Corporation is not affiliated with python-valve in any way.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index