diplomacy

Release 1.1.2

Jun 01, 2020

API Documentation

Diplomacy is a strategic board game when you play a country (power) on a map with the goal to conquer at least half to all the supply centers present on the map. To achieve this goal, you control power units (armies and/or fleets) that you can use to occupy empty provinces (locations), attack provinces occupied by other powers, or support other units occupying or attacking a position.

This is a complex game with many rules and corner cases to take into account, and, thus, an interesting subject for both entertainment (between humans) and studies (e.g. how to create an artificial intelligence good enough to beat humans). This project aims to provide a complete and working Python implementation of Diplomacy game with following features:

- A working game engine easy to use to get familiar with game rules, test corner cases, and simulate complete parties.
- An interface to allow the game to be played online, using:
  - a Python server implementation to handle many games
  - a Python client implementation to play remotely using all the power and facilities of Python
  - a web front-end to play remotely using a human user-friendly interface
- Some integration interface to play with other server/client implementations, especially:
  - a DAIDE server to play with DAIDE client bots
  - a webdiplomacy API to play with webdiplomacy server implementation

CHAPTER 1

diplomacy.client

1.1 diplomacy.client.channel

Channel

- The channel object represents an authenticated connection over a socket.
- It has a token that it sends with every request to authenticate itself.

class diplomacy.client.channel.**Channel**(*connection*, *token*)
Bases: object

Channel - Represents an authenticated connection over a physical socket

__init__(*connection*, *token*)
Initialize a channel.

Properties:

- **connection**: *Connection* object from which this channel originated.
- **token**: Channel token, used to identify channel on server.
- **game_id_to_instances**: Dictionary mapping a game ID to *NetworkGame* objects loaded for this game. Each *NetworkGame* has a specific role, which is either an observer role, an omniscient role, or a power (player) role. Network games for a specific game ID are managed within a GameInstancesSet, which makes sure that there will be at most 1 *NetworkGame* instance per possible role.

Parameters

- **connection** (diplomacy.client.connection.Connection) – a Connection object.
- **token** (*str*) – Channel token.

create_game(*game=None*, ***kwargs*)
Send request `CreateGame` with request parameters `kwargs`. Return response data returned by server for this request. See `CreateGame` about request parameters and response.

get_available_maps(*game=None*, ***kwargs*)
Send request `GetAvailableMaps` with request parameters `kwargs`. Return response data returned by server for this request. See `GetAvailableMaps` about request parameters and response.

get_playable_powers(*game=None*, ***kwargs*)
Send request `GetPlayablePowers` with request parameters `kwargs`. Return response data returned by server for this request. See `GetPlayablePowers` about request parameters and response.

join_game(*game=None*, ***kwargs*)
Send request `JoinGame` with request parameters `kwargs`. Return response data returned by server for this request. See `JoinGame` about request parameters and response.

join_powers(*game=None*, ***kwargs*)
Send request `JoinPowers` with request parameters `kwargs`. Return response data returned by server for this request. See `JoinPowers` about request parameters and response.

list_games(*game=None*, ***kwargs*)
Send request `ListGames` with request parameters `kwargs`. Return response data returned by server for this request. See `ListGames` about request parameters and response.

get_games_info(*game=None*, ***kwargs*)
Send request `GetGamesInfo` with request parameters `kwargs`. Return response data returned by server for this request. See `GetGamesInfo` about request parameters and response.

get_dummy_waiting_powers(*game=None*, ***kwargs*)
Send request `GetDummyWaitingPowers` with request parameters `kwargs`. Return response data returned by server for this request. See `GetDummyWaitingPowers` about request parameters and response.

delete_account(*game=None*, ***kwargs*)
Send request `DeleteAccount` with request parameters `kwargs`. Return response data returned by server for this request. See `DeleteAccount` about request parameters and response.

logout(*game=None*, ***kwargs*)
Send request `Logout` with request parameters `kwargs`. Return response data returned by server for this request. See `Logout` about request parameters and response.

make_omniscient(*game=None*, ***kwargs*)
Send request `SetGrade` with forced parameters `(grade=omniscient, grade_update=promote)` and additional request parameters `kwargs`. Return response data returned by server for this request. See `SetGrade` about request parameters and response.

remove_omniscient(*game=None*, ***kwargs*)
Send request `SetGrade` with forced parameters `(grade=omniscient, grade_update=demote)` and additional request parameters `kwargs`. Return response data returned by server for this request. See `SetGrade` about request parameters and response.

promote_administrator(*game=None*, ***kwargs*)
Send request `SetGrade` with forced parameters `(grade=admin, grade_update=promote)` and additional request parameters `kwargs`. Return response data returned by server for this request. See `SetGrade` about request parameters and response.

demote_administrator(*game=None*, ***kwargs*)
Send request `SetGrade` with forced parameters `(grade=admin, grade_update=demote)` and additional request parameters `kwargs`. Return response data returned by server for this request. See `SetGrade` about request parameters and response.

promote_moderator(*game=None*, ***kwargs*)
Send request *SetGrade* with forced parameters (grade=moderator, grade_update=promote) and additional request parameters kwargs. Return response data returned by server for this request. See *SetGrade* about request parameters and response.

demote_moderator(*game=None*, ***kwargs*)
Send request *SetGrade* with forced parameters (grade=moderator, grade_update=demote) and additional request parameters kwargs. Return response data returned by server for this request. See *SetGrade* about request parameters and response.

1.2 diplomacy.client.connection

Connection object, handling an internal websocket tornado connection.

diplomacy.client.connection.**connect**(*hostname*, *port*)
Connect to given hostname and port.

Parameters

- **hostname** (*str*) – a hostname
- **port** (*int*) – a port

Returns a Connection object connected.

Return type *Connection*

class diplomacy.client.connection.**Connection**(*hostname*, *port*, *use_ssl=False*)
Bases: object

Connection class.

The connection class should not be initiated directly, but through the connect method

```
>>> from diplomacy.client.connection import connect
>>> connection = await connect(hostname, port)
```

Properties:

- **hostname**: str hostname to connect (e.g. 'localhost')
- **port**: int port to connect (e.g. 8888)
- **use_ssl**: bool telling if connection should be securized (True) or not (False).
- **url**: (property) str websocket url to connect (generated with hostname and port)
- **connection**: tornado.websocket.WebSocketClientConnection a tornado websocket connection object
- **connection_count**: int number of successful connections from this Connection object. Used to check if message callbacks is already launched (if count > 0).
- **is_connecting**: tornado.locks.Event a tornado Event used to keep connection status. No request can be sent while is_connecting. If connected, Synchronize requests can be sent immediately even if is_reconnecting. Other requests must wait full reconnection.
- **is_reconnecting**: tornado.locks.Event a tornado Event used to keep re-connection status. Non-synchronize request cannot be sent while is_reconnecting. If reconnected, all requests can be sent.
- **channels**: a weakref.WeakValueDictionary mapping channel token to *Channel* object.

- **requests_to_send**: a `Dict` mapping a request ID to the context of a request **not sent**. If we are disconnected when trying to send a request, then request context is added to this dictionary to be send later once reconnected.
- **requests_waiting_responses**: a `Dict` mapping a request ID to the context of a request **sent**. Contains requests that are waiting for a server response.
- **unknown_tokens**: `set` a set of unknown tokens. We can safely ignore them, as the server has been notified.

__init__(*hostname*, *port*, *use_ssl=False*)
Constructor

The connection class should not be initiated directly, but through the connect method

```
>>> from diplomacy.client.connection import connect
>>> connection = await connect(hostname, port)
```

Parameters

- **hostname** (*str*) – hostname to connect (e.g. 'localhost')
- **port** (*int*) – port to connect (e.g. 8888)
- **use_ssl** (*bool*) – telling if connection should be securized (True) or not (False).

authenticate(*username*, *password*)
Send a *SignIn* request. User will be created on the server automatically if it doesn't exist.

Parameters

- **username** (*str*) – username
- **password** (*str*) – password

Returns a *Channel* object representing the authentication.

Return type *diplomacy.client.channel.Channel*

get_daide_port(*game_id*)
Send a *GetDaidePort* request.

Parameters **game_id** (*str*) – game id for which to retrieve the DAIDE port.

Returns the game DAIDE port

Return type int

1.3 diplomacy.client.network_game

Game object used on client side.

class diplomacy.client.network_game.**NetworkGame**(*channel*, *received_game*)
Bases: *diplomacy.engine.game.Game*

NetworkGame class.

Properties:

- **channel**: associated *diplomacy.client.channel.Channel* object.
- **notification_callbacks**: `Dict` mapping a notification class name to a callback to be called when a corresponding game notification is received.

__init__(*channel*, *received_game*)
Initialize network game object with a channel and a game object sent by server.

Parameters

- **channel** (diplomacy.client.channel.Channel) – a Channel object.
- **received_game** (diplomacy.engine.game.Game) – a Game object.

get_phase_history(***kwargs*)
Send game request *GetPhaseHistory* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *GetPhaseHistory* about request parameters and response.

leave(***kwargs*)
Send game request *LeaveGame* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *LeaveGame* about request parameters and response.

send_game_message(***kwargs*)
Send game request *SendGameMessage* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *SendGameMessage* about request parameters and response.

set_orders(***kwargs*)
Send game request *SetOrders* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *SetOrders* about request parameters and response.

clear_centers(***kwargs*)
Send game request *ClearCenters* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *ClearCenters* about request parameters and response.

clear_orders(***kwargs*)
Send game request *ClearOrders* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *ClearOrders* about request parameters and response.

clear_units(***kwargs*)
Send game request *ClearUnits* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *ClearUnits* about request parameters and response.

wait(***kwargs*)
Send game request *SetWaitFlag* with forced parameters `(wait=True)` and additional request parameters `kwargs`. See *SetWaitFlag* about request parameters and response.

no_wait(***kwargs*)
Send game request *SetWaitFlag* with forced parameters `(wait=False)` and additional request parameters `kwargs`. See *SetWaitFlag* about request parameters and response.

vote(***kwargs*)
Send game request *Vote* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *Vote* about request parameters and response.

save(***kwargs*)
Send game request *SaveGame* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *SaveGame* about request parameters and response.

synchronize()
Send a *Synchronize* request to synchronize this game with associated server game.

delete(***kwargs*)
Send game request *DeleteGame* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *DeleteGame* about request parameters and response.

kick_powers(***kwargs*)
Send game request *SetDummyPowers* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *SetDummyPowers* about request parameters and response.

set_state(***kwargs*)
Send game request *SetGameState* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *SetGameState* about request parameters and response.

process(***kwargs*)
Send game request *ProcessGame* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *ProcessGame* about request parameters and response.

query_schedule(***kwargs*)
Send game request *QuerySchedule* with forced parameters `( with request parameters )` and additional request parameters `kwargs`. See *QuerySchedule* about request parameters and response.

start(***kwargs*)
Send game request *SetGameStatus* with forced parameters `(status=active)` and additional request parameters `kwargs`. See *SetGameStatus* about request parameters and response.

pause(***kwargs*)
Send game request *SetGameStatus* with forced parameters `(status=paused)` and additional request parameters `kwargs`. See *SetGameStatus* about request parameters and response.

resume(***kwargs*)
Send game request *SetGameStatus* with forced parameters `(status=active)` and additional request parameters `kwargs`. See *SetGameStatus* about request parameters and response.

cancel(***kwargs*)
Send game request *SetGameStatus* with forced parameters `(status=canceled)` and additional request parameters `kwargs`. See *SetGameStatus* about request parameters and response.

draw(***kwargs*)
Send game request *SetGameStatus* with forced parameters `(status=completed)` and additional request parameters `kwargs`. See *SetGameStatus* about request parameters and response.

add_on_cleared_centers(*notification_callback*)
Add callback for notification *ClearedCenters*. Callback signature: `callback(network_game, notification) -> None`.

add_on_cleared_orders(*notification_callback*)
Add callback for notification *ClearedOrders*. Callback signature: `callback(network_game, notification) -> None`.

add_on_cleared_units(*notification_callback*)
Add callback for notification *ClearedUnits*. Callback signature: `callback(network_game, notification) -> None`.

add_on_game_deleted(*notification_callback*)
Add callback for notification *GameDeleted*. Callback signature: `callback(network_game, notification) -> None`.

add_on_game_message_received(*notification_callback*)
Add callback for notification *GameMessageReceived*. Callback signature: `callback(network_game, notification) -> None`.

add_on_game_processed(*notification_callback*)
Add callback for notification *GameProcessed*. Callback signature: `callback(network_game, notification) -> None`.

add_on_game_phase_update(*notification_callback*)
Add callback for notification *GamePhaseUpdate*. Callback signature: `callback(network_game, notification) -> None`.

add_on_game_status_update(*notification_callback*)
Add callback for notification *GameStatusUpdate*. Callback signature: `callback(network_game, notification) -> None`.

add_on_omniscient_updated(*notification_callback*)
Add callback for notification *OmniscientUpdated*. Callback signature: `callback(network_game, notification) -> None`.

add_on_power_orders_flag(*notification_callback*)
Add callback for notification *PowerOrdersFlag*. Callback signature: `callback(network_game, notification) -> None`.

add_on_power_orders_update(*notification_callback*)
Add callback for notification *PowerOrdersUpdate*. Callback signature: `callback(network_game, notification) -> None`.

add_on_power_vote_updated(*notification_callback*)
Add callback for notification *PowerVoteUpdated*. Callback signature: `callback(network_game, notification) -> None`.

add_on_power_wait_flag(*notification_callback*)
Add callback for notification *PowerWaitFlag*. Callback signature: `callback(network_game, notification) -> None`.

add_on_powers_controllers(*notification_callback*)
Add callback for notification *PowersControllers*. Callback signature: `callback(network_game, notification) -> None`.

add_on_vote_count_updated(*notification_callback*)
Add callback for notification *VoteCountUpdated*. Callback signature: `callback(network_game, notification) -> None`.

add_on_vote_updated(*notification_callback*)
Add callback for notification *VoteUpdated*. Callback signature: `callback(network_game, notification) -> None`.

clear_on_cleared_centers()
Clear callbacks for notification *ClearedCenters*..

clear_on_cleared_orders()
Clear callbacks for notification *ClearedOrders*..

clear_on_cleared_units()
Clear callbacks for notification *ClearedUnits*..

clear_on_game_deleted()
Clear callbacks for notification *GameDeleted*..

clear_on_game_message_received()
Clear callbacks for notification *GameMessageReceived*..

clear_on_game_processed()
Clear callbacks for notification *GameProcessed*..

clear_on_game_phase_update()
Clear callbacks for notification *GamePhaseUpdate*..

clear_on_game_status_update()
Clear callbacks for notification *GameStatusUpdate*..

clear_on_omniscient_updated()
Clear callbacks for notification *OmniscientUpdated*..

clear_on_power_orders_flag()
Clear callbacks for notification *PowerOrdersFlag*..

clear_on_power_orders_update()
Clear callbacks for notification *PowerOrdersUpdate*..

clear_on_power_vote_updated()
Clear callbacks for notification *PowerVoteUpdated*..

clear_on_power_wait_flag()
Clear callbacks for notification *PowerWaitFlag*..

clear_on_powers_controllers()
Clear callbacks for notification *PowersControllers*..

clear_on_vote_count_updated()
Clear callbacks for notification *VoteCountUpdated*..

clear_on_vote_updated()
Clear callbacks for notification *VoteUpdated*..

add_notification_callback(*notification_class*, *notification_callback*)
Add a callback for a notification.

Parameters

- **notification_class** – a notification class. See *diplomacy.communication.notifications* about available notifications.
- **notification_callback** – callback to add: `callback(network_game, notification) -> None`.

clear_notification_callbacks(*notification_class*)
Remove all user callbacks for a notification.

Parameters **notification_class** – a notification class

notify(*notification*)
Notify game with given notification (call associated callbacks if defined).

CHAPTER 2

diplomacy.communication

2.1 diplomacy.communication.notifications

Server -> Client notifications.

class diplomacy.communication.notifications.**AccountDeleted**(***kwargs*)
Bases: diplomacy.communication.notifications._ChannelNotification

Notification about an account deleted.

class diplomacy.communication.notifications.**OmniscientUpdated**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a grade updated. Sent at channel level.

Properties:

- **grade_update**: str One of 'promote' or 'demote'.
- **game**: parsing.JsonableClassType(Game) a *diplomacy.engine.game.Game* object.

class diplomacy.communication.notifications.**ClearedCenters**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about centers cleared.

class diplomacy.communication.notifications.**ClearedOrders**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about orders cleared.

class diplomacy.communication.notifications.**ClearedUnits**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about units cleared.

class diplomacy.communication.notifications.**VoteCountUpdated**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about new count of draw votes for a game (for observers).

Properties:

- **count_voted**: `int` number of powers that have voted.
- **count_expected**: `int` number of powers to be expected to vote.

class diplomacy.communication.notifications.**VoteUpdated**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about votes updated for a game (for omniscient observers).

Properties:

- **vote**: `Dict` mapping a power name to a Vote (`str`) object representing power vote. Possible votes are: yes, no, neutral.

class diplomacy.communication.notifications.**PowerVoteUpdated**(***kwargs*)
Bases: *diplomacy.communication.notifications.VoteCountUpdated*

Notification about a new vote for a specific game power (for player games).

Properties:

- **vote**: `str` vote object representing associated power vote. Can be yes, no, neutral.

class diplomacy.communication.notifications.**PowersControllers**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about current controller for each power in a game.

Properties:

- **powers**: A `Dict` that maps a power_name to a controller_name `str`.
- **timestamps**: A `Dict` that maps a power_name to timestamp where the controller took over.

class diplomacy.communication.notifications.**GameDeleted**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a game deleted.

class diplomacy.communication.notifications.**GameProcessed**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a game phase update. Sent after game has processed a phase.

Properties:

- **previous_phase_data**: `diplomacy.utils.game_phase_data.GamePhaseData` of the previous phase
- **current_phase_data**: `diplomacy.utils.game_phase_data.GamePhaseData` of the current phase

class diplomacy.communication.notifications.**GamePhaseUpdate**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a game phase update.

Properties:

- **phase_data**: `diplomacy.utils.game_phase_data.GamePhaseData` of the updated phase
- **phase_data_type**: `str`. One of 'state_history', 'state', 'phase'

class diplomacy.communication.notifications.**GameStatusUpdate**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a game status update.

Properties:

-**status**: `str`. One of 'forming', 'active', 'paused', 'completed', 'canceled'

class diplomacy.communication.notifications.**GameMessageReceived**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a game message received.

Properties:

- **message**: *diplomacy.engine.message.Message* received.

class diplomacy.communication.notifications.**PowerOrdersUpdate**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a power order update.

Properties:

- **orders**: List of updated orders (i.e. `str`)

class diplomacy.communication.notifications.**PowerOrdersFlag**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a power order flag update.

Properties:

- **order_is_set**: `int`. O = ORDER_NOT_SET, 1 = ORDER_SET_EMPTY, 2 = ORDER_SET.

class diplomacy.communication.notifications.**PowerWaitFlag**(***kwargs*)
Bases: diplomacy.communication.notifications._GameNotification

Notification about a power wait flag update.

Properties:

- **wait**: `bool` that indicates to wait until the deadline is reached before proceeding. Otherwise if all powers are not waiting, the game is processed as soon as all non-eliminated powers have submitted their orders.

diplomacy.communication.notifications.**parse_dict**(*json_notification*)
Parse a JSON expected to represent a notification. Raise an exception if parsing failed.

Parameters **json_notification** – JSON dictionary.

Returns a notification class instance.

2.2 diplomacy.communication.requests

Client -> Server requests.

This module contains the definition of request (as classes) that a client can send to Diplomacy server implemented in this project.

The client -> server communication follows this procedure:

- Client sends a request to server. All requests have parameters that must be filled by client before being sent.
- Server replies with a response, which is either an error response or a valid response.
- Client receives and handles server response.
  - If server response is an error, client converts it to a typed exception and raises it.

  - If server response is a valid response, client return either the response data directly, or make further treatments and return a derived data.

Diplomacy package actually provides 2 clients: the Python client and the web front-end.

Web front-end provides user-friendly forms to collect required request parameters, makes all request calls internally, and then uses them to update graphical user interface. So, when using front-end, you don't need to get familiar with underlying protocol, and documentation in this module won't be really useful for you.

Python client consists of three classes (*Connection*, *Channel* and *NetworkGame*) which provide appropriate methods to automatically send requests, handle server response, and either raise an exception (if server returns an error) or return a client-side wrapped data (if server returns a valid response) where requests were called. Thus, these methods still need to receive request parameters, and you need to know what kind of data they can return. So, if you use Python client, you will need the documentation in this module, which describes, for each request:

- the request parameters (important)
- the server valid responses (less interesting)
- the Python client returned values (important)

All requests classes inherit from `_AbstractRequest` which require parameters `name` (from parant class `NetworkData`), `request_id` and `re_sent`. These parameters are automatically filled by the client.

From parent class `_AbstractRequest`, we get 3 types of requests:

- public requests, which directly inherit from `_AbstractRequest`.
- channel requests, inherited from `_AbstractChannelRequest`, which requires additional parameter `token`. Token is retrieved by client when he connected to server using connection request *SignIn*, and is then used to create a *Channel* object. Channel object will be responsible for sending all other channel requests, automatically filling token field for these requests.
- game requests, intherited from `_AbstractGameRequest`, which itself inherit from `_AbstractChannelRequest`, and requires additional parameters `game_id`, `game_role` and `phase` (game short phase name). Game ID, role and phase are retrieved for a specific game by the client when he joined a game through one of featured *Channel* methods which return a *NetworkGame* object. Network game will then be responsible for sending all other game requests, automatically filling game ID, role and phase for these requests.

Then, all other requests derived directly from either abstract request class, abstract channel request class, or abstract game request class, may require additional parameters, and if so, these parameters will need to be filled by the user, by passing them to related client methods.

Check *Connection* for available public request methods (and associated requests).

Check *Channel* for available channel request methods (and associated requests).

Check *NetworkGame* for available game request methods (and associated requests).

Then come here to get parameters and returned values for associated requests.

class diplomacy.communication.requests.**GetDaidePort**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractRequest

Public request to get DAIDE port opened for a game.

Parameters **game_id** (*str*) – ID of game for which yu want to get DAIDE port

Returns

- Server: *DataPort*
- Client: int - DAIDE port

Raises *diplomacy.utils.exceptions.DaidePortException* – if there is no DAIDE port associated to given game ID.

class diplomacy.communication.requests.**SignIn**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractRequest

Connection request. Log in or sign in to server.

Parameters

- **username** (*str*) – account username
- **password** (*str*) – account password

Returns

- Server: *DataToken*
- Client: a *Channel* object presenting user connected to the server. If any sign in error occurs, raise an appropriate *ResponseException*.

class diplomacy.communication.requests.**CreateGame**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to create a game.

Parameters

- **game_id** (*str, optional*) – game ID. If not provided, a game ID will be generated.
- **n_controls** (*int, optional*) – number of controlled powers required to start the game. A power becomes controlled when a player joins the game to control this power. Game won't start as long it does not have this number of controlled powers. Game will stop (to `forming` state) if the number of controlled powers decrease under this number (e.g. when powers are kicked, eliminated, or when a player controlling a power leaves the game). If not provided, set with the number of powers on the map (e.g. `7` on standard map).
- **deadline** (*int, optional*) – (default `300`) time (in seconds) for the game to wait before processing a phase. `0` means no deadline, ie. game won't process a phase until either all powers submit orders and turn off wait flag, or a game master forces game to process.
- **registration_password** (*str, optional*) – password required to join the game. If not provided, anyone can join the game.
- **power_name** (*str, optional*) – power to control once game is created.
  - If provided, the user who send this request will be joined to the game as a player controlling this power.
  - If not provided, the user who send this request will be joined to the game as an omniscient observer (ie. able to see everything in the game, including user messages). Plus, as game creator, user will also be a game master, ie. able to send master requests, e.g. to force game processing.
- **state** (*dict, optional*) – game initial state (for expert users).
- **map_name** (*str, optional*) – (default `'standard'`) map to play on. You can retrieve maps available on server by sending request *GetAvailableMaps*.
- **rules** (*list, optional*) – list of strings - game rules (for expert users).

Returns

- Server: *DataGame*

- Client: a *NetworkGame* object representing a client version of the game created and joined. Either a power game (if power name given) or an omniscient game.

class diplomacy.communication.requests.**DeleteAccount**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to delete an account.

Parameters **username** (*str, optional*) – name of user to delete account

- if **not** given, then account to delete will be the one of user sending this request.
- if **provided**, then user submitting this request must have administrator privileges.

Returns None

class diplomacy.communication.requests.**GetDummyWaitingPowers**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to get games with dummy waiting powers. A dummy waiting power is a dummy (not controlled) power:

- not yet eliminated,
- without orders submitted (for current game phase),
- but able to submit orders (for current game phase),
- and who is waiting for orders.

It's a non-controlled orderable free power, which is then best suited to be controlled by an automated player (e.g. a bot, or a learning algorithm).

Parameters **buffer_size** (*int*) – maximum number of powers to return.

Returns

- Server: *DataGamesToPowerNames*
- Client: a dictionary mapping a game ID to a list of dummy waiting power names, such that the total number of power names in the entire dictionary does not exceed given buffer size.

class diplomacy.communication.requests.**GetAvailableMaps**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to get maps available on server.

Returns

- Server: *DataMaps*
- Client: a dictionary associating a map name to a dictionary of information related to the map. You can especially check key `'powers'` to get the list of map power names.

class diplomacy.communication.requests.**GetPlayablePowers**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to get the list of playable powers for a game. A playable power is a dummy (uncontrolled) power not yet eliminated.

Parameters **game_id** (*str*) – ID of game to get playable powers

Returns

- Server: *DataPowerNames*
- Client: set of playable power names for given game ID.

class diplomacy.communication.requests.**JoinGame**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to join a game.

Parameters

- **game_id** (*str*) – ID of game to join
- **power_name** (*str, optional*) – if provided, name of power to control. Otherwise, user wants to observe game without playing.
- **registration_password** (*str, optional*) – password to join game. If omitted while game requires a password, server will return an error.

Returns

- Server: *DataGame*
- Client: a *NetworkGame* object representing the client game, which is either:
  - a power game (if power name was given), meaning that this network game allows user to play a power
  - an observer game, if power was not given and user does not have omniscient privileges for this game. Observer role allows user to watch game phases changes, orders submitted and orders results for each phase, but he can not see user messages and he can not send any request that requires game master privileges.
  - an omniscient game, if power was not given and user does have game master privileges. Omniscient role allows user to see everything in the game, including user messages. If user does only have omniscient privileges for this game, he can't do anything more, If he does have up to game master privileges, then he can also send requests that require game master privileges.

class diplomacy.communication.requests.**JoinPowers**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to join many powers of a game with one request.

This request is mostly identical to *JoinGame*, except that list of power names is mandatory. It's useful to allow the user to control many powers while still working with 1 client game instance.

Parameters

- **game_id** (*str*) – ID of game to join
- **power_names** (*list, optional*) – list of power names to join
- **registration_password** (*str, optionl*) – password to join the game

Returns None. If request succeeds, then the user is registered as player for all given power names. The user can then simply join game to one of these powers (by sending a *JoinGame* request), and he will be able to manage all the powers through the client game returned by *JoinGame*.

class diplomacy.communication.requests.**ListGames**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to find games.

Parameters

- **game_id** (*str, optional*) – if provided, look for games with game ID either containing or contained into this game ID.
- **status** (*str, optional*) – if provided, look for games with this status.

- **map_name** (*str, optional*) – if provided, look for games with this map name.
- **include_protected** (*bool optional*) – (default True) tell if we must look into games protected by a password
- **for_omniscience** (*bool, optional*) – (default False) tell if we look for games where request user can be at least omniscient.

Returns

- Server: *DataGames*
- Client: a list of *DataGameInfo* objects, each containing a bunch of information about a game found. If no game found, list will be empty.

class diplomacy.communication.requests.**GetGamesInfo**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to get information for a given list of game indices.

Parameters **games** (*list*) – list of game ID.

Returns

- Server: *DataGames*
- Client: a list of *DataGameInfo* objects.

class diplomacy.communication.requests.**Logout**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to logout. Returns nothing.

class diplomacy.communication.requests.**UnknownToken**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to tell server that a channel token is unknown.

Note: Client does not even wait for a server response when sending this request, which acts more like a "client notification" sent to server.

---

class diplomacy.communication.requests.**SetGrade**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractChannelRequest

Channel request to modify the grade of a user. Require admin privileges to change admin grade, and at least game master privileges to change omniscient or moderator grade.

Parameters

- **grade** (*str*) – grade to update (`'omniscient'`, `'admin'` or `'moderator'`)
- **grade_update** (*str*) – how to make update (`'promote'` or `'demote'`)
- **username** (*str*) – user for which the grade must be modified
- **game_id** (*str, optional*) – ID of game for which the grade must be modified. Required only for `'moderator'` and `'omniscient'` grade.

Returns None

class diplomacy.communication.requests.**ClearCenters**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to clear supply centers. See method *Game.clear_centers()*.

Parameters **power_name** (*str, optional*) – if given, clear centers for this power. Otherwise, clear centers for all powers.

Returns None

class diplomacy.communication.requests.**ClearOrders**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to clear orders.

Parameters **power_name** (*str, optional*) – if given, clear orders for this power. Otherwise, clear orders for all powers.

Returns None

class diplomacy.communication.requests.**ClearUnits**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to clear units.

Parameters **power_name** (*str, optional*) – if given, clear units for this power. Otherwise, clear units for all powers.

Returns None

class diplomacy.communication.requests.**DeleteGame**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to delete a game. Require game master privileges. Returns nothing.

class diplomacy.communication.requests.**GetAllPossibleOrders**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to get all possible orders. Return (server and client) a *DataPossibleOrders* object containing possible orders and orderable locations.

class diplomacy.communication.requests.**GetPhaseHistory**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to get a list of game phase data from game history for given phases interval. A phase can be either None, a phase name (string) or a phase index (integer). See *Game.get_phase_history()* about how phases are used to retrieve game phase data.

Parameters

- **from_phase** (*str | int, optional*) – phase from which to look in game history
- **to_phase** (*str | int, optional*) – phase up to which to look in game history

Returns

- Server: DataGamePhases
- Client: a list of GamePhaseData objects corresponding to game phases found between from_phase and to_phase in game history.

class diplomacy.communication.requests.**LeaveGame**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to leave a game (logout from game). If request power name is set (ie. request user was a player), then power will become uncontrolled. Otherwise, user will be signed out from its observer (or omniscient) role. Returns nothing.

class diplomacy.communication.requests.**ProcessGame**(**kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to force a game processing. Require master privileges. Return nothing.

class diplomacy.communication.requests.**QuerySchedule**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to get info about current scheduling for a game in server. Returns (server and client) a *DataGameSchedule* object.

class diplomacy.communication.requests.**SaveGame**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to get game exported in JSON format.

Returns

- Server: *DataSavedGame*
- Client: dict - the JSON dictionary.

class diplomacy.communication.requests.**SendGameMessage**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game message to send a user request.

Parameters **message** (Message) – message to send. See *Message* for more info. message sender must be request user role (ie. power role, in such case). Message time sent must not be defined, it will be allocated by server.

Returns

- Server: *DataTimeStamp*
- Client: nothing (returned timestamp is just used to update message locally)

class diplomacy.communication.requests.**SetDummyPowers**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to set dummy powers. Require game master privileges. If given powers are controlled, related players are kicked and powers become dummy (uncontrolled).

Parameters

- **power_names** (*list, optional*) – list of power names to set dummy. If not provided, will be all map power names.
- **username** – if provided, only power names controlled by this user will be set dummy.

Returns None

class diplomacy.communication.requests.**SetGameState**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to set a game state (for exper users). Require game master privileges.

Parameters

- **state** (*dict*) – game state
- **orders** (*dict*) – dictionary mapping a power name to a list of orders strings
- **results** (*dict*) – dictionary mapping a unit to a list of order result strings
- **messages** (*dict*) – dictionary mapping a timestamp to a message

Returns None

class diplomacy.communication.requests.**SetGameStatus**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to force game status (only if new status differs from previous one). Require game master privileges.

Parameters **status** (*str*) – game status to set. Either `'forming'`, `'active'`, `'paused'`, `'completed'` or `'canceled'`.

- If new status is `'completed'`, game will be forced to draw.
- If new status is `'active'`, game will be forced to start.
- If new status is `'paused'`, game will be forced to pause.
- If new status is `'canceled'`, game will be canceled and become invalid.

Returns None

class diplomacy.communication.requests.**SetOrders**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to set orders for a power.

Parameters

- **power_name** (*str, optional*) – power name. If not given, request user must be a game player, and power is inferred from request game role.
- **orders** (*list*) – list of power orders.
- **wait** (*bool, optional*) – if provided, wait flag to set for this power.

Returns None

class diplomacy.communication.requests.**SetWaitFlag**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to set orders for a power.

Parameters

- **power_name** (*str, optional*) – power name. If not given, request user must be a game player, and power if inferred from request game role.
- **wait** (*bool*) – wait flag to set.

Returns None

class diplomacy.communication.requests.**Synchronize**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to force synchronization of client game with server game. If necessary, server will send appropriate notifications to client game so that it can be up to date with server game state.

Parameters **timestamp** (*int*) – timestamp since which client game needs to synchronize.

Returns (server and client) a *DataGameInfo* object.

class diplomacy.communication.requests.**Vote**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Game request to vote for draw decision. If number of pro-draw votes > number of con-draw votes for current phase, then server will automatically draw the game and send appropriate notifications. Votes are reset after a game processing.

Parameters

- **power_name** (*str, optional*) – power name who wants to vote. If not provided, request user must be a game player, and power name will be inferred from request game role.
- **vote** (*str*) – vote to set. Either `'yes'` (power votes for draw), `'no'` (power votes against draw), or `'neutral'` (power does not want to decide).

Returns None

`diplomacy.communication.requests.`**`parse_dict`**(*json_request*)

Parse a JSON dictionary expected to represent a request. Raise an exception if parsing failed.

Parameters **json_request** (*dict*) – JSON dictionary.

Returns a request class instance.

Return type _AbstractRequest | _AbstractChannelRequest | _AbstractGameRequest

2.3 diplomacy.communication.responses

Server -> Client responses sent by server as replies to requests.

class `diplomacy.communication.responses.`**`Error`**(***kwargs*)

Bases: `diplomacy.communication.responses._AbstractResponse`

Error response sent when an error occurred on server-side while handling a request.

Properties:

- **error_type**: str - error type, containing the exception class name.
- **message**: str - error message

throw()

Convert this error to an instance of a Diplomacy ResponseException class and raises it.

class `diplomacy.communication.responses.`**`Ok`**(***kwargs*)

Bases: `diplomacy.communication.responses._AbstractResponse`

Ok response sent by default after handling a request. Contains nothing.

class `diplomacy.communication.responses.`**`NoResponse`**(***kwargs*)

Bases: `diplomacy.communication.responses._AbstractResponse`

Placeholder response to indicate that no responses are required

class `diplomacy.communication.responses.`**`DataGameSchedule`**(***kwargs*)

Bases: `diplomacy.communication.responses._AbstractResponse`

Response with info about current scheduling for a game.

Properties:

- **game_id**: str - game ID
- **phase**: str - game phase
- **schedule**: `SchedulerEvent` - scheduling information about the game

class `diplomacy.communication.responses.`**`DataGameInfo`**(***kwargs*)

Bases: `diplomacy.communication.responses._AbstractResponse`

Response containing information about a game, to be used when no entire game object is required.

Properties:

- **game_id**: game ID
- **phase**: game phase
- **timestamp**: latest timestamp when data was saved into game on server (ie. game state or message)
- **timestamp_created**: timestamp when game was created on server
- **map_name**: (optional) game map name
- **observer_level**: (optional) highest observer level allowed for the user who sends the request. Either `'observer_type'`, `'omniscient_type'` or `'master_type'`.
- **controlled_powers**: (optional) list of power names controlled by the user who sends the request.
- **rules**: (optional) game rules
- **status**: (optional) game status
- **n_players**: (optional) number of powers currently controlled in the game
- **n_controls**: (optional) number of controlled powers required by the game to be active
- **deadline**: (optional) game deadline - time to wait before processing a game phase
- **registration_password**: (optional) boolean - if True, a password is required to join the game

class diplomacy.communication.responses.**DataPossibleOrders**(***kwargs*)
Bases: diplomacy.communication.responses._AbstractResponse

Response containing information about possible orders for a game at its current phase.

Properties:

- **possible_orders**: dictionary mapping a location short name to all possible orders here
- **orderable_locations**: dictionary mapping a power name to its orderable locations

class diplomacy.communication.responses.**UniqueData**(***kwargs*)
Bases: diplomacy.communication.responses._AbstractResponse

Response containing only 1 field named `data`. A derived class will contain a specific typed value in this field.

classmethod validate_params()
Called when getting model to validate parameters. Called once per class.

class diplomacy.communication.responses.**DataToken**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a token.

class diplomacy.communication.responses.**DataMaps**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing maps info (dictionary mapping a map name to a dictionary with map information).

class diplomacy.communication.responses.**DataPowerNames**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a list of power names.

class diplomacy.communication.responses.**DataGames**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a list of *DataGameInfo* objects.

class diplomacy.communication.responses.**DataPort**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a DAIDE port (integer).

class diplomacy.communication.responses.**DataTimeStamp**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a timestamp (integer).

class diplomacy.communication.responses.**DataGamePhases**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a list of `GamePhaseData` objects.

class diplomacy.communication.responses.**DataGame**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a *Game* object.

class diplomacy.communication.responses.**DataSavedGame**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a game saved in JSON dictionary.

class diplomacy.communication.responses.**DataGamesToPowerNames**(***kwargs*)
Bases: *diplomacy.communication.responses.UniqueData*

Unique data containing a dictionary mapping a game ID to a list of power names.

diplomacy.communication.responses.**parse_dict**(*json_response*)
Parse a JSON dictionary expected to represent a response. Raise an exception if either:

- parsing failed
- response received is an Error response. In such case, a ResponseException is raised with the error message.

Parameters **json_response** – a JSON dict.

Returns a Response class instance.

CHAPTER 3

diplomacy.daide

3.1 diplomacy.daide.notifications

DAIDE Notifications - Contains a list of responses sent by the server to the client

class diplomacy.daide.notifications.**DaideNotification**(***kwargs*)
Bases: object

Represents a DAIDE response.

__init__(***kwargs*)
Constructor

to_bytes()
Returning the bytes representation of the response

to_string()
Returning the string representation of the response

class diplomacy.daide.notifications.**MapNameNotification**(*map_name*, ***kwargs*)
Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a MAP DAIDE response. Sends the name of the current map to the client.

Syntax:

```
MAP ('name')
```

__init__(*map_name*, ***kwargs*)
Builds the response :param map_name: String. The name of the current map.

class diplomacy.daide.notifications.**HelloNotification**(*power_name*, *passcode*, *level*, *deadline*, *rules*, ***kwargs*)
Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a HLO DAIDE response. Sends the power to be played by the client with the passcode to rejoin the game and the details of the game.

Syntax:

```
HLO (power) (passcode) (variant) (variant) ...
```

Variant syntax:

```
LVL n           # Level of the syntax accepted
MTL seconds     # Movement time limit
RTL seconds     # Retreat time limit
BTL seconds     # Build time limit
DSD             # Disables the time limit when a client disconects
AOA             # Any orders accepted
```

LVL 10:

Variant syntax:

```
PDA             # Accept partial draws
NPR             # No press during retreat phases
NPB             # No press during build phases
PTL seconds     # Press time limit
```

__init__(*power_name*, *passcode*, *level*, *deadline*, *rules*, ***kwargs*)
Builds the response

Parameters

- **power_name** – The name of the power being played.
- **passcode** – Integer. A passcode to rejoin the game.
- **level** – Integer. The daide syntax level of the game
- **deadline** – Integer. The number of seconds per turn (0 to disable)
- **rules** – The list of game rules.

class diplomacy.daide.notifications.**SupplyCenterNotification**(*powers_centers*, *map_name*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a SCO DAIDE notification. Sends the current supply centre ownership.

Syntax:

```
SCO (power centre centre ...) (power centre centre ...) ...
```

__init__(*powers_centers*, *map_name*, ***kwargs*)
Builds the notification

Parameters

- **powers_centers** – A dict of {power_name: centers} objects
- **map_name** – The name of the map

class diplomacy.daide.notifications.**CurrentPositionNotification**(*phase_name*, *powers_units*, *powers_retreats*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a NOW DAIDE notification. Sends the current turn, and the current unit positions.

Syntax:

```
NOW (turn) (unit) (unit) ...
```

Unit syntax:

```
power unit_type province
power unit_type province MRT (province province ...)
```

__init__(*phase_name*, *powers_units*, *powers_retreats*, ***kwargs*)
Builds the notification

Parameters

- **phase_name** – The name of the current phase (e.g. 'S1901M')
- **powers** – A list of *diplomacy.engine.power.Power* objects

class diplomacy.daide.notifications.**MissingOrdersNotification**(*phase_name*, *power*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a MIS DAIDE response. Sends the list of unit for which an order is missing or indication about required disbands or builds.

Syntax:

```
MIS (unit) (unit) ...
MIS (unit MRT (province province ...)) (unit MRT (province province ...)) ...
MIS (number)
```

__init__(*phase_name*, *power*, ***kwargs*)
Builds the response :param phase_name: The name of the current phase (e.g. 'S1901M') :param power: The power to check for missing orders :type power: diplomacy.engine.power.Power

class diplomacy.daide.notifications.**OrderResultNotification**(*phase_name*, *order_bytes*, *results*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a ORD DAIDE response. Sends the result of an order after the turn has been processed.

Syntax:

```
ORD (turn) (order) (result)
ORD (turn) (order) (result RET)
```

Result syntax:

```
SUC          # Order succeeded (can apply to any order).
BNC          # Move bounced (only for MTO, CTO or RTO orders).
CUT          # Support cut (only for SUP orders).
DSR          # Move via convoy failed due to dislodged convoying fleet (only for␣
↪CTO orders).
NSO          # No such order (only for SUP, CVY or CTO orders).
RET          # Unit was dislodged and must retreat.
```

__init__(*phase_name*, *order_bytes*, *results*, ***kwargs*)
Builds the response

Parameters

- **phase_name** – The name of the current phase (e.g. 'S1901M')
- **order_bytes** – The bytes received for the order
- **results** – An array containing the error codes.

class diplomacy.daide.notifications.**TimeToDeadlineNotification**(*seconds*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a TME DAIDE response. Sends the time to the next deadline.

Syntax:

```
TME (seconds)
```

__init__(*seconds*, ***kwargs*)

Builds the response :param seconds: Integer. The number of seconds before deadline

class diplomacy.daide.notifications.**PowerInCivilDisorderNotification**(*power_name*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a CCD DAIDE response. Sends the name of the power in civil disorder.

Syntax:

```
CCD (power)
```

__init__(*power_name*, ***kwargs*)

Builds the response :param power_name: The name of the power being played.

class diplomacy.daide.notifications.**PowerIsEliminatedNotification**(*power_name*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a OUT DAIDE response. Sends the name of the power eliminated.

Syntax:

```
OUT (power)
```

__init__(*power_name*, ***kwargs*)

Builds the response :param power_name: The name of the power being played.

class diplomacy.daide.notifications.**DrawNotification**(***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a DRW DAIDE response. Indicates that the game has ended due to a draw

Syntax:

```
DRW
```

__init__(***kwargs*)

Builds the response

class diplomacy.daide.notifications.**MessageFromNotification**(*from_power_name*, *to_power_names*, *message*, ***kwargs*)

Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a FRM DAIDE response. Indicates that the game has ended due to a draw

Syntax:

```
FRM (power) (power power ...) (press_message)
FRM (power) (power power ...) (reply)
```

__init__(*from_power_name*, *to_power_names*, *message*, ***kwargs*)
Builds the response

class diplomacy.daide.notifications.**SoloNotification**(*power_name*, ***kwargs*)
Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a SLO DAIDE response. Indicates that the game has ended due to a solo by the specified power

Syntax:

```
SLO (power)
```

__init__(*power_name*, ***kwargs*)
Builds the response :param power_name: The name of the power being solo.

class diplomacy.daide.notifications.**SummaryNotification**(*phase_name*, *powers*, *daide_users*, *years_of_elimination*, ***kwargs*)
Bases: *diplomacy.daide.notifications.DaideNotification*

Represents a SMR DAIDE response. Sends the summary for each power at the end of the game

Syntax:

```
SMR (turn) (power_summary) ...
```

power_summary syntax:

```
power ('name') ('version') number_of_centres
power ('name') ('version') number_of_centres year_of_elimination
```

__init__(*phase_name*, *powers*, *daide_users*, *years_of_elimination*, ***kwargs*)
Builds the Notification

class diplomacy.daide.notifications.**TurnOffNotification**(***kwargs*)
Bases: *diplomacy.daide.notifications.DaideNotification*

Represents an OFF DAIDE response. Requests a client to exit

Syntax:

```
OFF
```

__init__(***kwargs*)
Builds the response

diplomacy.daide.notifications.**MAP**
alias of *diplomacy.daide.notifications.MapNameNotification*

diplomacy.daide.notifications.**HLO**
alias of *diplomacy.daide.notifications.HelloNotification*

diplomacy.daide.notifications.**SCO**
alias of *diplomacy.daide.notifications.SupplyCenterNotification*

diplomacy.daide.notifications.**NOW**
alias of *diplomacy.daide.notifications.CurrentPositionNotification*

diplomacy.daide.notifications.**MIS**
alias of *diplomacy.daide.notifications.MissingOrdersNotification*

diplomacy.daide.notifications.**ORD**
alias of *diplomacy.daide.notifications.OrderResultNotification*

diplomacy.daide.notifications.**TME**
alias of *diplomacy.daide.notifications.TimeToDeadlineNotification*

diplomacy.daide.notifications.**CCD**
alias of *diplomacy.daide.notifications.PowerInCivilDisorderNotification*

diplomacy.daide.notifications.**OUT**
alias of *diplomacy.daide.notifications.PowerIsEliminatedNotification*

diplomacy.daide.notifications.**DRW**
alias of *diplomacy.daide.notifications.DrawNotification*

diplomacy.daide.notifications.**FRM**
alias of *diplomacy.daide.notifications.MessageFromNotification*

diplomacy.daide.notifications.**SLO**
alias of *diplomacy.daide.notifications.SoloNotification*

diplomacy.daide.notifications.**SMR**
alias of *diplomacy.daide.notifications.SummaryNotification*

diplomacy.daide.notifications.**OFF**
alias of *diplomacy.daide.notifications.TurnOffNotification*

3.2 diplomacy.daide.requests

Daide Requests - Contains a list of requests sent by client to server

class diplomacy.daide.requests.**RequestBuilder**
Bases: object

Builds DaideRequest from bytes or tokens

static from_bytes(*daide_bytes*, ***kwargs*)
Builds a request from DAIDE bytes

Parameters **daide_bytes** – The bytes representation of a request

Returns The DaideRequest built from the bytes

class diplomacy.daide.requests.**DaideRequest**(***kwargs*)
Bases: diplomacy.communication.requests._AbstractGameRequest

Represents a DAIDE request.

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**NameRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a NME DAIDE request. Can be sent by the client as soon as it connects to the server.

Syntax:

```
NME ('name') ('version')
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**ObserverRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a NME DAIDE request. Can be sent by the client as soon as it connects to the server.

Syntax:

```
OBS
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**IAmRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a IAM DAIDE request. Can be sent by the client at anytime to rejoin the game.

Syntax:

```
IAM (power) (passcode)
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**HelloRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a HLO DAIDE request. Sent by the client to request a copy of the HLO message.

Syntax:

```
HLO
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**MapRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a MAP DAIDE request. Sent by the client to request a copy of the MAP message.

Syntax:

```
MAP
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**MapDefinitionRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a MDF DAIDE request. Sent by the client to request the map definition of the game.

Syntax:

```
MDF
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**SupplyCentreOwnershipRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a SCO DAIDE request. Sent by the client to request a copy of the last SCO message.

Syntax:

```
SCO
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**CurrentPositionRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a NOW DAIDE request. Sent by the client to request a copy of the last NOW message.

Syntax:

```
NOW
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**HistoryRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a HST DAIDE request. Sent by the client to request a copy of a previous ORD, SCO and NOW messages.

Syntax:

```
HST (turn)
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**SubmitOrdersRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a SUB DAIDE request. Sent by the client to submit orders.

Syntax:

```
SUB (order) (order) ...
SUB (turn) (order) (order) ...
```

order syntax:

```
(unit) HLD                                          # Hold
(unit) MTO province                                 # Move to
(unit) SUP (unit)                                   # Support
(unit) SUP (unit) MTO (prov_no_coast)               # Support to move
(unit) CVY (unit) CTO province                      # Convoy
(unit) CTO province VIA (sea_prov sea_prov ...)     # Convoy to via provinces
(unit) RTO province                                 # Retreat to
(unit) DSB                                          # Disband (R phase)
(unit) BLD                                          # Build
(unit) REM                                          # Remove (A phase)
(unit) WVE                                          # Waive
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**MissingOrdersRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a MIS DAIDE request. Sent by the client to request a copy of the current MIS message.

Syntax:

```
MIS
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**GoFlagRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a GOF DAIDE request. Sent by the client to notify that the client is ready to process the turn.

Syntax:

```
GOF
```

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**TimeToDeadlineRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a TME DAIDE request. Sent by the client to request a TME message or to request it at a later time.

Syntax:

```
TME
TME (seconds)
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**DrawRequest**(**kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a DRW DAIDE request. Sent by the client to notify that the client would accept a draw.

Syntax:

```
DRW
```

LVL 10:

```
DRW (power power ...)
```

__init__(**kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**SendMessageRequest**(**kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a SND DAIDE request

Syntax:

```
SND (power ...) (press_message)
SND (power ...) (reply)
SND (turn) (power ...) (press_message)
SND (turn) (power ...) (reply)
```

Press message syntax:

```
PRP (arrangement)
CCL (press_message)
FCT (arrangement)
TRY (tokens)
```

Reply syntax:

```
YES (press_message)
REJ (press_message)
BWX (press_message)
HUH (press_message)
```

__init__(**kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**NotRequest**(**kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a NOT DAIDE request. Sent by the client to cancel a previous request.

Syntax:

```
NOT (SUB)                          # Cancel all submitted orders
NOT (SUB (order))                  # Cancel specific submitted order
NOT (GOF)                          # Do not process orders until the deadline
NOT (TME)                          # Cancel all requested time messages
NOT (TME (seconds))                # Cancel specific requested time message
NOT (DRW)                          # Cancel all draw requests
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**AcceptRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a YES DAIDE request.

Syntax:

```
YES (MAP ('name'))
YES (SVE ('gamename'))
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**RejectRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a REJ DAIDE request.

Syntax:

```
REJ (SVE ('gamename'))
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**ParenthesisErrorRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a PRN DAIDE request. Sent by the client to specify an error in the set of parenthesis.

Syntax:

```
PRN (message)
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**SyntaxErrorRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a HUH DAIDE request. Sent by the client to specify an error in a message.

Syntax:

```
HUH (message)
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

class diplomacy.daide.requests.**AdminMessageRequest**(***kwargs*)
Bases: *diplomacy.daide.requests.DaideRequest*

Represents a ADM DAIDE request. Can be sent by the client to send a message to all clients. Should not be used for negotiation.

Syntax:

```
ADM ('message')
```

__init__(***kwargs*)
Constructor

parse_bytes(*daide_bytes*)
Builds the request from DAIDE bytes

diplomacy.daide.requests.**NME**
alias of *diplomacy.daide.requests.NameRequest*

diplomacy.daide.requests.**OBS**
alias of *diplomacy.daide.requests.ObserverRequest*

diplomacy.daide.requests.**IAM**
alias of *diplomacy.daide.requests.IAmRequest*

diplomacy.daide.requests.**HLO**
alias of *diplomacy.daide.requests.HelloRequest*

diplomacy.daide.requests.**MAP**
alias of *diplomacy.daide.requests.MapRequest*

diplomacy.daide.requests.**MDF**
alias of *diplomacy.daide.requests.MapDefinitionRequest*

diplomacy.daide.requests.**SCO**
alias of *diplomacy.daide.requests.SupplyCentreOwnershipRequest*

diplomacy.daide.requests.**NOW**
alias of *diplomacy.daide.requests.CurrentPositionRequest*

diplomacy.daide.requests.**HST**
alias of *diplomacy.daide.requests.HistoryRequest*

diplomacy.daide.requests.**SUB**
alias of *diplomacy.daide.requests.SubmitOrdersRequest*

diplomacy.daide.requests.**MIS**
alias of *diplomacy.daide.requests.MissingOrdersRequest*

diplomacy.daide.requests.**GOF**
alias of *diplomacy.daide.requests.GoFlagRequest*

diplomacy.daide.requests.**TME**
alias of *diplomacy.daide.requests.TimeToDeadlineRequest*

diplomacy.daide.requests.**DRW**
alias of *diplomacy.daide.requests.DrawRequest*

diplomacy.daide.requests.**SND**
alias of *diplomacy.daide.requests.SendMessageRequest*

diplomacy.daide.requests.**NOT**
alias of *diplomacy.daide.requests.NotRequest*

diplomacy.daide.requests.**YES**
alias of *diplomacy.daide.requests.AcceptRequest*

diplomacy.daide.requests.**REJ**
alias of *diplomacy.daide.requests.RejectRequest*

diplomacy.daide.requests.**PRN**
alias of *diplomacy.daide.requests.ParenthesisErrorRequest*

diplomacy.daide.requests.**HUH**
alias of *diplomacy.daide.requests.SyntaxErrorRequest*

diplomacy.daide.requests.**ADM**
alias of *diplomacy.daide.requests.AdminMessageRequest*

3.3 diplomacy.daide.responses

DAIDE Responses - Contains a list of responses sent by the server to the client

class diplomacy.daide.responses.**DaideResponse**(***kwargs*)
Bases: diplomacy.communication.responses._AbstractResponse

Represents a DAIDE response.

__init__(***kwargs*)
Constructor

class diplomacy.daide.responses.**MapNameResponse**(*map_name*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a MAP DAIDE response. Sends the name of the current map to the client.

Syntax:

```
MAP ('name')
```

__init__(*map_name*, ***kwargs*)
Builds the response :param map_name: String. The name of the current map.

class diplomacy.daide.responses.**MapDefinitionResponse**(*map_name*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a MDF DAIDE response. Sends configuration of a map to a client

Syntax:

```
MDF (powers) (provinces) (adjacencies)
```

powers syntax:

```
power power ...
```

power syntax:

```
AUS                     # Austria
ENG                     # England
FRA                     # France
GER                     # Germany
ITA                     # Italy
RUS                     # Russia
TUR                     # Turkey
```

provinces syntax:

```
(supply_centres) (non_supply_centres)
```

supply_centres syntax:

```
(power centre centre ...) (power centre centre ...) ...
```

supply_centres power syntax:

```
(power power ...)       # This is currently not supported
AUS                     # Austria
ENG                     # England
FRA                     # France
GER                     # Germany
ITA                     # Italy
RUS                     # Russia
TUR                     # Turkey
UNO                     # Unknown power
```

non_supply_centres syntax:

```
province province ...   # List of provinces
```

adjacencies syntax:

```
(prov_adjacencies) (prov_adjacencies) ...
```

prov_adjacencies syntax:

```
province (unit_type adjacent_prov adjacent_prov ...) (unit_type adjacent_prov␣
↪adjacent_prov ...) ...
```

unit_type syntax:

```
AMY                     # List of provinces an army can move to
FLT                     # List of provinces a fleet can move to
(FLT coast)             # List of provinces a fleet can move to from the given␣
↪coast
```

adjacent_prov syntax:

```
province                # A province which can be moved to
(province coast)        # A coast of a province that can be moved to
```

__init__(*map_name*, ***kwargs*)
Builds the response

Parameters **map_name** – The name of the map

class diplomacy.daide.responses.**HelloResponse**(*power_name*, *passcode*, *level*, *deadline*, *rules*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a HLO DAIDE response. Sends the power to be played by the client with the passcode to rejoin the game and the details of the game.

Syntax:

```
HLO (power) (passcode) (variant) (variant) ...
```

Variant syntax:

```
LVL n           # Level of the syntax accepted
MTL seconds     # Movement time limit
RTL seconds     # Retreat time limit
BTL seconds     # Build time limit
DSD             # Disables the time limit when a client disconects
AOA             # Any orders accepted
```

LVL 10:

Variant syntax:

```
PDA             # Accept partial draws
NPR             # No press during retreat phases
NPB             # No press during build phases
PTL seconds     # Press time limit
```

__init__(*power_name*, *passcode*, *level*, *deadline*, *rules*, ***kwargs*)
Builds the response

Parameters

- **power_name** – The name of the power being played.
- **passcode** – Integer. A passcode to rejoin the game.
- **level** – Integer. The daide syntax level of the game
- **deadline** – Integer. The number of seconds per turn (0 to disable)
- **rules** – The list of game rules.

class diplomacy.daide.responses.**SupplyCenterResponse**(*powers_centers*, *map_name*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a SCO DAIDE response. Sends the current supply centre ownership.

Syntax:

```
SCO (power centre centre ...) (power centre centre ...) ...
```

__init__(*powers_centers*, *map_name*, ***kwargs*)
Builds the response

Parameters

- **powers_centers** – A dict of {power_name: centers} objects
- **map_name** – The name of the map

class diplomacy.daide.responses.**CurrentPositionResponse**(*phase_name*, *powers_units*, *powers_retreats*, ***kwargs*)

Bases: *diplomacy.daide.responses.DaideResponse*

Represents a NOW DAIDE response. Sends the current turn, and the current unit positions.

Syntax:

```
NOW (turn) (unit) (unit) ...
```

Unit syntax:

```
power unit_type province
power unit_type province MRT (province province ...)
```

__init__(*phase_name*, *powers_units*, *powers_retreats*, ***kwargs*)

Builds the response

Parameters

- **phase_name** – The name of the current phase (e.g. 'S1901M')
- **powers** – A list of *diplomacy.engine.power.Power* objects

class diplomacy.daide.responses.**ThanksResponse**(*order_bytes*, *results*, ***kwargs*)

Bases: *diplomacy.daide.responses.DaideResponse*

Represents a THX DAIDE response. Sends the result of an order after submission.

Syntax:

```
THX (order) (note)
```

Note syntax:

```
MBV     # Order is OK.
FAR     # Not adjacent.
NSP     # No such province
NSU     # No such unit
NAS     # Not at sea (for a convoying fleet)
NSF     # No such fleet (in VIA section of CTO or the unit performing a CVY)
NSA     # No such army (for unit being ordered to CTO or for unit being CVYed)
NYU     # Not your unit
NRN     # No retreat needed for this unit
NVR     # Not a valid retreat space
YSC     # Not your supply centre
ESC     # Not an empty supply centre
HSC     # Not a home supply centre
NSC     # Not a supply centre
CST     # No coast specified for fleet build in StP, or an attempt
          to build a fleet inland, or an army at sea.
NMB     # No more builds allowed
NMR     # No more removals allowed
NRS     # Not the right season
```

__init__(*order_bytes*, *results*, ***kwargs*)

Builds the response

Parameters

- **order_bytes** – The bytes received for the order
- **results** – An array containing the error codes.

class diplomacy.daide.responses.**MissingOrdersResponse**(*phase_name*, *power*, ***kwargs*)

Bases: *diplomacy.daide.responses.DaideResponse*

Represents a MIS DAIDE response. Sends the list of unit for which an order is missing or indication about required disbands or builds.

Syntax:

```
MIS (unit) (unit) ...
MIS (unit MRT (province province ...)) (unit MRT (province province ...)) ...
MIS (number)
```

__init__(*phase_name*, *power*, ***kwargs*)

Builds the response

Parameters

- **phase_name** – The name of the current phase (e.g. 'S1901M')
- **power** (diplomacy.engine.power.Power) – The power to check for missing orders

class diplomacy.daide.responses.**OrderResultResponse**(*phase_name*, *order_bytes*, *results*, ***kwargs*)

Bases: *diplomacy.daide.responses.DaideResponse*

Represents a ORD DAIDE response. Sends the result of an order after the turn has been processed.

Syntax:

```
ORD (turn) (order) (result)
ORD (turn) (order) (result RET)
```

Result syntax:

```
SUC          # Order succeeded (can apply to any order).
BNC          # Move bounced (only for MTO, CTO or RTO orders).
CUT          # Support cut (only for SUP orders).
DSR          # Move via convoy failed due to dislodged convoying fleet (only for␣
↪CTO orders).
NSO          # No such order (only for SUP, CVY or CTO orders).
RET          # Unit was dislodged and must retreat.
```

__init__(*phase_name*, *order_bytes*, *results*, ***kwargs*)

Builds the response

Parameters

- **phase_name** – The name of the current phase (e.g. 'S1901M')
- **order_bytes** – The bytes received for the order
- **results** – An array containing the error codes.

class diplomacy.daide.responses.**TimeToDeadlineResponse**(*seconds*, ***kwargs*)

Bases: *diplomacy.daide.responses.DaideResponse*

Represents a TME DAIDE response. Sends the time to the next deadline.

Syntax:

```
TME (seconds)
```

__init__(*seconds*, ***kwargs*)
Builds the response

Parameters **seconds** – Integer. The number of seconds before deadline

class diplomacy.daide.responses.**AcceptResponse**(*request_bytes*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a YES DAIDE request.

Syntax:

```
YES (TME (seconds))                                  # Accepts to set the time␣
↪when a
                                                       TME message will be sent
YES (NOT (TME))                                      # Accepts to cancel all␣
↪requested time messages
YES (NOT (TME (seconds)))                            # Accepts to cancel a␣
↪specific requested time message
YES (GOF)                                            # Accepts to wait until the␣
↪deadline before processing
                                                       the orders for the turn
YES (NOT (GOF))                                      # Accepts to cancel to wait␣
↪until the deadline before
                                                       processing the orders for␣
↪the turn
YES (DRW)                                            # Accepts to draw
YES (NOT (DRW))                                      # Accepts to cancel a draw␣
↪request
```

LVL 10:

```
YES (DRW (power power ...))                          # Accepts a partial draw
YES (NOT (DRW (power power ...)))                    # Accepts to cancel a partial␣
↪draw request
                                                       (? not mentinned in the␣
↪DAIDE doc)
YES (SND (power power ...) (press_message))          # Accepts a press message
YES (SND (turn) (power power ...) (press_message))   # Accepts a press message
```

__init__(*request_bytes*, ***kwargs*)
Builds the response

Parameters **request_bytes** – The bytes received for the request

class diplomacy.daide.responses.**RejectResponse**(*request_bytes*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a REJ DAIDE request.

Syntax:

```
REJ (NME ('name') ('version'))                       # Rejects a client in the game
REJ (IAM (power) (passcode))                         # Rejects a client to rejoin␣
↪the game
```

(continues on next page)

(continued from previous page)

```
REJ (HLO)                                          # Rejects to send the HLO␣
↪message
REJ (HST (turn))                                   # Rejects to send a copy of a␣
↪previous
                                                     ORD, SCO and NOW messages
REJ (SUB (order) (order))                          # Rejects a submition of␣
↪orders
REJ (SUB (turn) (order) (order))                   # Rejects a submition of␣
↪orders
REJ (NOT (SUB (order)))                            # Rejects a cancellation of a␣
↪submitted order
REJ (MIS)                                          # Rejects to send a copy of␣
↪the current MIS message
REJ (GOF)                                          # Rejects to wait until the␣
↪deadline before processing
                                                     the orders for the turn
REJ (NOT (GOF))                                    # Rejects to cancel to wait␣
↪until the deadline before
                                                     processing the orders for␣
↪the turn
REJ (TME (seconds))                                # Rejects to set the time␣
↪when a
                                                     TME message will be sent
REJ (NOT (TME))                                    # Rejects to cancel all␣
↪requested time messages
REJ (NOT (TME (seconds)))                          # Rejects to cancel a␣
↪specific requested time message
REJ (ADM ('name') ('message')                      # Rejects the admin message
REJ (DRW)                                          # Rejects to draw
REJ (NOT (DRW))                                    # Rejects to cancel a draw␣
↪request
```

LVL 10:

```
REJ (DRW (power power ...))                        # Rejects to partially draw
REJ (NOT (DRW (power power ...)))                  # Rejects to cancel a partial␣
↪draw request
REJ (SND (power power ...) (press_message))        # Rejects a press message
REJ (SND (turn) (power power ...) (press_message)) # Rejects a press message
```

__init__(*request_bytes*, ***kwargs*)
Builds the response

Parameters **request_bytes** – The bytes received for the request

class diplomacy.daide.responses.**NotResponse**(*response_bytes*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a NOT DAIDE response.

Syntax:

```
NOT (CCD (power))
```

__init__(*response_bytes*, ***kwargs*)
Builds the response :param response_bytes: The bytes received for the request

class diplomacy.daide.responses.**PowerInCivilDisorderResponse**(*power_name*, ***kwargs*)

Bases: *diplomacy.daide.responses.DaideResponse*

Represents a CCD DAIDE response. Sends the name of the power in civil disorder.

Syntax:

```
CCD (power)
```

__init__(*power_name*, ***kwargs*)
Builds the response

Parameters **power_name** – The name of the power being played.

class diplomacy.daide.responses.**PowerIsEliminatedResponse**(*power_name*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a OUT DAIDE response. Sends the name of the power eliminated.

Syntax:

```
OUT (power)
```

__init__(*power_name*, ***kwargs*)
Builds the response

Parameters **power_name** – The name of the power being played.

class diplomacy.daide.responses.**ParenthesisErrorResponse**(*request_bytes*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a PRN DAIDE response.

Syntax:

```
PRN (message)
```

__init__(*request_bytes*, ***kwargs*)
Builds the response

Parameters **request_bytes** – The bytes received for the request

class diplomacy.daide.responses.**SyntaxErrorResponse**(*request_bytes*, *error_index*, ***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents a HUH DAIDE response.

Syntax:

```
HUH (message)
```

__init__(*request_bytes*, *error_index*, ***kwargs*)
Builds the response

Parameters

- **request_bytes** – The bytes received for the request
- **error_index** – The index of the faulty token

class diplomacy.daide.responses.**TurnOffResponse**(***kwargs*)
Bases: *diplomacy.daide.responses.DaideResponse*

Represents an OFF DAIDE response. Requests a client to exit

Syntax:

```
OFF
```

__init__(***kwargs*)
Builds the response

diplomacy.daide.responses.**MAP**
alias of *diplomacy.daide.responses.MapNameResponse*

diplomacy.daide.responses.**MDF**
alias of *diplomacy.daide.responses.MapDefinitionResponse*

diplomacy.daide.responses.**HLO**
alias of *diplomacy.daide.responses.HelloResponse*

diplomacy.daide.responses.**SCO**
alias of *diplomacy.daide.responses.SupplyCenterResponse*

diplomacy.daide.responses.**NOW**
alias of *diplomacy.daide.responses.CurrentPositionResponse*

diplomacy.daide.responses.**THX**
alias of *diplomacy.daide.responses.ThanksResponse*

diplomacy.daide.responses.**MIS**
alias of *diplomacy.daide.responses.MissingOrdersResponse*

diplomacy.daide.responses.**ORD**
alias of *diplomacy.daide.responses.OrderResultResponse*

diplomacy.daide.responses.**TME**
alias of *diplomacy.daide.responses.TimeToDeadlineResponse*

diplomacy.daide.responses.**YES**
alias of *diplomacy.daide.responses.AcceptResponse*

diplomacy.daide.responses.**REJ**
alias of *diplomacy.daide.responses.RejectResponse*

diplomacy.daide.responses.**NOT**
alias of *diplomacy.daide.responses.NotResponse*

diplomacy.daide.responses.**CCD**
alias of *diplomacy.daide.responses.PowerInCivilDisorderResponse*

diplomacy.daide.responses.**OUT**
alias of *diplomacy.daide.responses.PowerIsEliminatedResponse*

diplomacy.daide.responses.**OFF**
alias of *diplomacy.daide.responses.TurnOffResponse*

diplomacy.daide.responses.**PRN**
alias of *diplomacy.daide.responses.ParenthesisErrorResponse*

diplomacy.daide.responses.**HUH**
alias of *diplomacy.daide.responses.SyntaxErrorResponse*

CHAPTER 4

diplomacy.engine

4.1 diplomacy.engine.game

Game

- Contains the game engine

class diplomacy.engine.game.**Game**(*game_id=None*, ***kwargs*)
Bases: diplomacy.utils.jsonable.Jsonable

Game class.

Properties:

- **combat**:
  - Dictionary of dictionaries containing the strength of every attack on a location (including units who don't count toward dislodgment)
  - Format: {loc: attack_strength: [['src loc', [support loc]]}
  - e.g. `{ 'MUN': { 1 : [ ['A MUN', [] ], ['A RUH', [] ] ], 2 : [ ['A SIL', ['A BOH'] ] ] } }`. MUN is holding, being attack without support from RUH and being attacked with support from SIL (S from BOH)
- **command**: contains the list of finalized orders to be processed (same format as orders, but without .order). e.g. {'A PAR': '- A MAR'}
- **controlled_powers**: *(for client games only)*. List of powers currently controlled by associated client user.
- **convoy_paths**:
  - Contains the list of remaining convoys path for each convoyed unit to reach their destination
  - Note: This is used to see if there are still active convoy paths remaining.
  - Note: This also include the start and ending location
  - e.g. {'A PAR': [['PAR', 'ION','NAO', 'MAR], ['PAR', 'ION', 'MAR']], ... }

- **convoy_paths_possible**:
  - Contains the list of possible convoy paths given the current fleet locations or None
  - e.g. [(START_LOC, {Fleets Req}, {possible dest}), ...]
- **convoy_paths_dest**:
  - Contains a dictionary of possible paths to reach destination from start or None
  - e.g. {start_loc: {dest_loc_1: [{fleets}, {fleets}, {fleets}], dest_loc_2: [{fleets, fleets}]}
- **daide_port**: *(for client games only)*. Port when a DAIDE bot can connect, to play with this game.
- **deadline**: integer: game deadline in seconds.
- **dislodged**: contains a dictionary of dislodged units (and the site that dislodged them'). e.g. { 'A PAR': 'MAR' }
- **error**: contains a list of errors that the game generated. e.g. ['NO MASTER SPECIFIED']
- **fixed_state**:
  - used when game is a context of a with-block.
  - Store values that define the game state when entered in with-statement.
  - Compared to actual fixed state to detect any changes in methods where changes are not allowed.
  - Reset to None when exited from with-statement.
- **game_id**: String that contains the current game's ID. e.g. '123456'
- **lost**:
  - Contains a dictionary of centers that have been lost during the term
  - e.g. {'PAR': 'FRANCE'} to indicate that PAR was lost by France (previous owner)
- **map**: Contains a reference to the current map (Map instance). e.g. map = Map('standard')
- **map_name**: Contains a reference to the name of the map that was loaded (or a path to a custom map file) e.g. map_name = 'standard' or map_name = '/some/path/to/file.map'
- **messages** *(for non-observer games only)*:
  - history of messages exchanged inside this game.
  - Sorted dict mapping message timestamps to message objects (instances of diplomacy.Message).
  - Format: {message.time_sent => message}
- **message_history**:
  - history of messages through all played phases.
  - Sorted dict mapping a short phase name to a message dict (with same format as field *message* describe above).
  - Format: {short phase name => {message.time_sent => message}}
  - Wrapped in a sorted dict at runtime, see method __init__().
- **meta_rules**: contains the rules that have been processed as directives. e.g. ['NO_PRESS']
- **n_controls**: integer:
  - exact number of controlled powers allowed for this game.

  - If game start mode is not START_MASTER, then game starts as soon as this number of powers are controlled.
- **no_rules**: contains the list of rules that have been disabled (prefixed with '!'). e.g ['NO_PRESS']
- **note**: a note to display on the rendering. e.g. 'Winner: FRANCE'
- **observer_level** *(for client games only)*:
  - Highest observation level allowed for associated client user.
  - Either "master_type", "omniscient_type" or "observer_type".
- **orders**: contains the list of current orders (not yet processed). e.g. {'A PAR': '- A MAR'}
- **ordered_units**:
  - Contains a dictionary of the units ordered by each power in the last phase
  - e.g. {'FRANCE': ['A PAR', 'A MAR'], 'ENGLAND': ... }
- **order_history**:
  - Contains the history of orders from each player from the beginning of the game.
  - Sorted dict mapping a short phase name to a dictionary of orders (powers names as keys, powers orders as values).
  - Format: {short phase name => {power name => [orders]}}
  - Wrapped in a sorted dict at runtime, see method __init__().
- **outcome**: contains the game outcome. e.g. [lastPhase, victor1, victor2, victor3]
- **phase**: string that contains a long representation of the current phase. e.g. 'SPRING 1901 MOVEMENT'
- **phase_type**: indicates the current phase type. e.g. 'M' for Movement, 'R' for Retreats, 'A' for Adjustment, '-' for non-playing phase
- **popped**: contains a list of all retreaters who didn't make it. e.g. ['A PAR', 'A MAR']
- **powers**:
  - Contains a dictionary mapping power names to power instances in the game
  - e.g. {'FRANCE': FrancePower, 'ENGLAND': EnglishPower, ... }
- **registration_password**: ** hashed ** version of password to be sent by a player to join this game.
- **renderer**: contains the object in charge of rendering the map. e.g. Renderer()
- **result**:
  - Contains the result of the action for each unit.
  - In Movement Phase, result can be 'no convoy', 'bounce', 'void', 'cut', 'dislodged', 'disrupted'. e.g. { 'A PAR': ['cut', 'void'] }
  - In Retreats phase, result can be 'bounce', 'disband', 'void'. e.g. { 'A PAR': ['cut', 'void'] }
  - In Adjustments phase, result can be 'void' or ''. e.g. { 'A PAR': ['', 'void'] } # e.g. to indicate a successful build, and a void build.
- **result_history**:
  - Contains the history of orders results for all played phases.
  - Sorted dict mapping a short phase name to a dictionary of order results for this phase.
  - Dictionary of order results maps a unit to a list of results. See field result for more details.

  - Format: {short phase name => {unit => [results]}}
  - Wrapped in a sorted dict at runtime, see method __init__().
- **role**: Either a power name (for player game) or a value in diplomacy.utils.strings.ALL_ROLE_TYPES.
- **rules**: Contains a list of active rules. e.g. ['NO_PRESS', ...]. Default is `diplomacy.utils.constants.DEFAULT_GAME_RULES`.
- **state_history**:
  - history of previous game states (returned by method get_state()) for this game.
  - Sorted dict mapping a short phase name to a game state.
  - Each game state is associated to a timestamp generated when state is created by method get_state().
  - State timestamp then represents the "end" time of the state, ie. time when this state was saved and archived in state history.
  - Format: {short phase name => state}
  - Wrapped in a sorted dict at runtime, see method __init__().
- **status**: game status (forming, active, paused, completed or canceled). Possible values in diplomacy.utils.strings.ALL_GAME_STATUSES.
- **supports**:
  - Contains a dictionary of support for each unit
  - Format: { 'unit': [nb_of_support, [list of supporting units]] }
  - e.g. { 'A PAR': [2, ['A MAR']] }. 2 support, but the Marseille support does NOT count toward dislodgment
- **timestamp_created**: timestamp in microseconds when game object was created on server side.
- **victory**:
  - Indicates the number of SUPPLY [default] centers one power must control to win the game
  - Format: [reqFirstYear, reqSecondYear, ..., reqAllFurtherYears]
  - e.g. [10,10,18] for 10 the 1st year, 10 the 2nd year, 18 year 3+
- **win** - Indicates the minimum number of centers required to win. e.g. 3
- **zobrist_hash** - Contains the zobrist hash representing the current state of this game. e.g. 12545212418541325

Cache properties:

- **unit_owner_cache**:
  - Contains a dictionary with (unit, coast_required) as key and owner as value
  - Set to Note when the cache is not built
  - e.g. {('A PAR', True): <FRANCE>, ('A PAR', False): <FRANCE>), ... }

__init__(*game_id=None*, ***kwargs*)
Constructor

power
(only for player games) Return client power associated to this game.

Returns a Power object.

Return type *diplomacy.engine.power.Power*

is_game_done
Returns a boolean flag that indicates if the game is done

current_state()
Returns the game object. To be used with the following syntax:

```
with game.current_state():
    orders = players.get_orders(game, power_name)
    game.set_orders(power_name, orders)
```

is_fixed_state_unchanged(*log_error=True*)
Check if actual state matches saved fixed state, if game is used as context of a with-block.

Parameters **log_error** – Boolean that indicates to log an error if state has changed

Returns boolean that indicates if the state has changed.

is_player_game()
Return True if this game is a player game.

is_observer_game()
Return True if this game is an observer game.

is_omniscient_game()
Return True if this game is an omniscient game.

is_server_game()
Return True if this game is a server game.

is_valid_password(*registration_password*)
Return True if given plain password matches registration password.

is_controlled(*power_name*)
Return True if given power name is currently controlled.

Parameters **power_name** (*str*) – power name

Return type bool

is_dummy(*power_name*)
Return True if given power name is not currently controlled.

does_not_wait()
Return True if the game does not wait anything to process its current phase. The game is not waiting is all **controlled** powers have defined orders and wait flag set to False. If it's a solitaire game (with no controlled powers), all (dummy, not eliminated) powers must have defined orders and wait flag set to False. By default, wait flag for a dummy power is True. Note that an empty orders set is considered as a defined order as long as it was explicitly set by the power controller.

has_power(*power_name*)
Return True if this game has given power name.

has_expected_controls_count()
Return True if game has expected number of map powers to be controlled. If True, the game can start (if not yet started).

count_controlled_powers()
Return the number of controlled map powers.

get_controlled_power_names(*username*)
Return the list of power names currently controlled by given user name.

get_expected_controls_count()
Return the number of map powers expected to be controlled in this game. This number is either specified in settings or the number of map powers.

get_dummy_power_names()
Return sequence of not eliminated dummy power names.

get_dummy_unordered_power_names()
Return a sequence of playable dummy power names without orders but still orderable and with orderable locations.

get_controllers()
Return a dictionary mapping each power name to its current controller name.

get_controllers_timestamps()
Return a dictionary mapping each power name to its controller timestamp.

get_random_power_name()
Return a random power name from remaining dummy power names. Raise an exception if there are no dummy power names.

get_latest_timestamp()
Return timestamp of latest data saved into this game (either current state, archived state or message).

Returns a timestamp

Return type int

classmethod filter_messages(*messages*, *game_role*, *timestamp_from=None*, *timestamp_to=None*)
Filter given messages based on given game role between given timestamps (bounds included). See method diplomacy.utils.SortedDict.sub() about bound rules.

Parameters

- **messages** (*diplomacy.utils.sorted_dict.SortedDict*) – a sorted dictionary of messages to filter.
- **game_role** – game role requiring messages. Either a special power name (PowerName.OBSERVER or PowerName.OMNISCIENT), a power name, or a list of power names.
- **timestamp_from** – lower timestamp (included) for required messages.
- **timestamp_to** – upper timestamp (included) for required messages.

Returns a dict of corresponding messages (empty if no corresponding messages found), mapping messages timestamps to messages.

get_phase_history(*from_phase=None*, *to_phase=None*, *game_role=None*)
Return a list of game phase data from game history between given phases (bounds included). Each GamePhaseData object contains game state, messages, orders and order results for a phase.

Parameters

- **from_phase** – either:
  - a string: phase name
  - an integer: index of phase in game history
  - None (default): lowest phase stored in game history
- **to_phase** – either:
  - a string: phase name

- an integer: index of phase in game history
- None (default): latest phase stored in game history

- **game_role** – (optional) role of game for which phase history is retrieved. If none, messages in game history will not be filtered.

Returns a list of GamePhaseHistory objects

get_phase_from_history(*short_phase_name*, *game_role=None*)
Return a game phase data corresponding to given phase from phase history.

phase_history_from_timestamp(*timestamp*)
Return list of game phase data from game history for which state timestamp >= given timestamp.

extend_phase_history(*game_phase_data*)
Add data from a game phase to game history.

Parameters **game_phase_data** (*GamePhaseData*) – a GamePhaseData object.

set_status(*status*)
Set game status with given status (should be in diplomacy.utils.strings.ALL_GAME_STATUSES).

draw(*winners=None*)
Force a draw for this game, set status as COMPLETED and finish the game.

Parameters **winners** – (optional) either None (all powers remaining to map are considered winners) or a sequence of required power names to be considered as winners.

Returns a couple (previous state, current state) with game state before the draw and game state after the draw.

set_controlled(*power_name*, *username*)
Control power with given username (may be None to set dummy power). See method diplomacy.Power#set_controlled.

update_dummy_powers(*dummy_power_names*)
Force all power associated to given dummy power names to be uncontrolled.

Parameters **dummy_power_names** – Sequence of required dummy power names.

update_powers_controllers(*powers_controllers*, *timestamps*)
Update powers controllers.

Parameters

- **powers_controllers** (*dict*) – a dictionary mapping a power name to a controller name.
- **timestamps** – a dictionary mapping a power name to timestamp when related controller (in powers_controllers) was associated to power.

new_power_message(*recipient*, *body*)
Create a undated (without timestamp) power message to be sent from a power to another via server. Server will answer with timestamp, and message will be updated and added to local game messages.

Parameters

- **recipient** – recipient power name (string).
- **body** – message body (string).

Returns a new GameMessage object.

Return type GameMessage

new_global_message(*body*)
Create an undated (without timestamp) global message to be sent from a power via server. Server will answer with timestamp, and message will be updated and added to local game messages.

Parameters **body** – message body (string).

Returns a new GameMessage object.

Return type *Message*

add_message(*message*)
Add message to current game data. Only a server game can add a message with no timestamp: game will auto-generate a timestamp for the message.

Parameters **message** – a GameMessage object to add.

Returns message timestamp.

Return type int

has_draw_vote()
Return True if all controlled non-eliminated powers have voted YES to draw game at current phase.

count_voted()
Return the count of controlled powers who already voted for a draw for current phase.

clear_vote()
Clear current vote.

get_map_power_names()
Return sequence of map power names.

get_current_phase()
Returns the current phase (format 'S1901M' or 'FORMING' or 'COMPLETED')

get_units(*power_name=None*)
Retrieves the list of units for a power or for all powers

Parameters **power_name** – Optional. The name of the power (e.g. `'FRANCE'`) or None for all powers

Returns A list of units (e.g. `['A PAR', 'A MAR']`) if a power name is provided or a dictionary of powers with their units if None is provided (e.g. `{'FRANCE': [...], ...}`)

Note: Dislodged units will appear with a leading asterisk (e.g. `'*A PAR'`)

get_centers(*power_name=None*)
Retrieves the list of owned supply centers for a power or for all powers

Parameters **power_name** – Optional. The name of the power (e.g. 'FRANCE') or None for all powers

Returns A list of supply centers (e.g. ['PAR', 'MAR']) if a power name is provided or a dictionary of powers with their supply centers if None is provided (e.g. {'FRANCE': [...], ...}

get_orders(*power_name=None*)
Retrieves the orders submitted by a specific power, or by all powers

Parameters **power_name** – Optional. The name of the power (e.g. 'FRANCE') or None for all powers

Returns A list of orders (e.g. ['A PAR H', 'A MAR - BUR']) if a power name is provided or a dictionary of powers with their orders if None is provided (e.g. {'FRANCE': ['A PAR H', 'A MAR - BUR', ...], ...}

get_orderable_locations(*power_name=None*)
Find the location requiring an order for a power (or for all powers)

Parameters **power_name** – Optionally, the name of the power (e.g. 'FRANCE') or None for all powers

Returns A list of orderable locations (e.g. ['PAR', 'MAR']) if a power name is provided or a dictionary of powers with their orderable locations if None is not provided (e.g. {'FRANCE': [...], ...}

get_order_status(*power_name=None*, *unit=None*, *loc=None*)
Returns a list or a dict representing the order status ('', 'no convoy', 'bounce', 'void', 'cut', 'dislodged', 'disrupted') for orders submitted in the last phase

Parameters

- **power_name** – Optional. If provided (e.g. 'FRANCE') will only return the order status of that power's orders
- **unit** – Optional. If provided (e.g. 'A PAR') will only return that specific unit order status.
- **loc** – Optional. If provided (e.g. 'PAR') will only return that specific loc order status. Mutually exclusive with unit
- **phase_type** – Optional. Returns the results of a specific phase type (e.g. 'M', 'R', or 'A')

Returns

- If unit is provided a list (e.g. [] or ['void', 'dislodged'])
- If loc is provided, a couple of unit and list (e.g. ('A PAR', ['void', 'dislodged'])), or (loc, []) if unit not found.
- If power is provided a dict (e.g. {'A PAR': ['void'], 'A MAR': []})
- Otherwise a 2-level dict (e.g. {'FRANCE: {'A PAR': ['void'], 'A MAR': []}, 'ENGLAND': {}, ... }

get_power(*power_name*)
Retrieves a power instance from given power name.

Parameters **power_name** – name of power instance to retrieve. Power name must be as given in map file.

Returns the power instance, or None if power name is not found.

Return type *diplomacy.engine.power.Power*

set_units(*power_name*, *units*, *reset=False*)
Sets units directly on the map

Parameters

- **power_name** – The name of the power who will own the units (e.g. 'FRANCE')
- **units** – An unit (e.g. 'A PAR') or a list of units (e.g. ['A PAR', 'A MAR']) to set Note units starting with a '*' will be set as dislodged
- **reset** – Boolean. If, clear all units of the power before setting them

Returns Nothing

set_centers(*power_name*, *centers*, *reset=False*)
Transfers supply centers ownership

Parameters

- **power_name** – The name of the power who will control the supply centers (e.g. 'FRANCE')
- **centers** – A loc (e.g. 'PAR') or a list of locations (e.g. ['PAR', 'MAR']) to transfer
- **reset** – Boolean. If, removes ownership of all power's SC before transferring ownership of the new SC

Returns Nothing

set_orders(*power_name*, *orders*, *expand=True*, *replace=True*)
Sets the current orders for a power

Parameters

- **power_name** – The name of the power (e.g. 'FRANCE')
- **orders** – The list of orders (e.g. ['A MAR - PAR', 'A PAR - BER', ...])
- **expand** – Boolean. If set, performs order expansion and reformatting (e.g. adding unit type, etc.) If false, expect orders in the following format. False gives a performance improvement.
- **replace** – Boolean. If set, replace previous orders on same units, otherwise prevents re-orders.

Returns Nothing

Expected format:

```
A LON H, F IRI - MAO, A IRI - MAO VIA, A WAL S F LON, A WAL S F MAO - IRI,
F NWG C A NWY - EDI, A IRO R MAO, A IRO D, A LON B, F LIV B
```

set_wait(*power_name*, *wait*)
Set wait flag for a power.

Parameters

- **power_name** – name of power to set wait flag.
- **wait** – wait flag (boolean).

clear_units(*power_name=None*)
Clear the power's units

Parameters **power_name** – Optional. The name of the power whose units will be cleared (e.g. 'FRANCE'), otherwise all units on the map will be cleared

Returns Nothing

clear_centers(*power_name=None*)
Removes ownership of supply centers

Parameters **power_name** – Optional. The name of the power whose centers will be cleared (e.g. 'FRANCE'), otherwise all centers on the map will lose ownership.

Returns Nothing

clear_orders(*power_name=None*)
Clears the power's orders

Parameters **power_name** – Optional. The name of the power to clear (e.g. 'FRANCE') or will clear orders for all powers if None.

Returns Nothing

clear_cache()
Clears all caches

set_current_phase(*new_phase*)
Changes the phase to the specified new phase (e.g. 'S1901M')

render(*incl_orders=True*, *incl_abbrev=False*, *output_format='svg'*, *output_path=None*)
Renders the current game and returns its image representation

Parameters

- **incl_orders** (*bool, optional*) – Optional. Flag to indicate we also want to render orders.
- **incl_abbrev** (*bool, optional*) – Optional. Flag to indicate we also want to display the provinces abbreviations.
- **output_format** (*str, optional*) – The desired output format. Currently, only 'svg' is supported.
- **output_path** (*str | None, optional*) – Optional. The full path where to save the rendering on disk.

Returns The rendered image in the specified format.

add_rule(*rule*)
Adds a rule to the current rule list

Parameters **rule** – Name of rule to add (e.g. 'NO_PRESS')

Returns Nothing

remove_rule(*rule*)
Removes a rule from the current rule list

Parameters **rule** – Name of rule to remove (e.g. 'NO_PRESS')

Returns Nothing

load_map(*reinit_powers=True*)
Load a map and process directives

Parameters **reinit_powers** – Boolean. If true, empty powers dict.

Returns Nothing, but stores the map in self.map

process()
Processes the current phase of the game.

Returns game phase data with data before processing.

build_caches()
Rebuilds the various caches

rebuild_hash()
Completely recalculate the Zobrist hash

Returns The updated hash value

get_hash()
Returns the zobrist hash for the current game

update_hash(*power*, *unit_type=''*, *loc=''*, *is_dislodged=False*, *is_center=False*, *is_home=False*)
Updates the zobrist hash for the current game

Parameters

- **power** – The name of the power owning the unit, supply center or home
- **unit_type** – Contains the unit type of the unit being added or remove from the board ('A' or 'F')
- **loc** – Contains the location of the unit, supply center, of home being added or remove
- **is_dislodged** – Indicates that the unit being added/removed is dislodged
- **is_center** – Indicates that the location being added/removed is a supply center
- **is_home** – Indicates that the location being added/removed is a home

Returns Nothing

get_phase_data()
Return a GamePhaseData object representing current game.

set_phase_data(*phase_data*, *clear_history=True*)
Set game from phase data.

Parameters

- **phase_data** – either a GamePhaseData or a list of GamePhaseData. If phase_data is a GamePhaseData, it will be treated as a list of GamePhaseData with 1 element. Last phase data in given list will be used to set current game internal state. Previous phase data in given list will replace current game history.
- **clear_history** – Indicate if we must clear game history fields before update.

get_state()
Gets the internal saved state of the game. This state is intended to represent current game view (powers states, orders results for previous phase, and few more info). See field message_history to get messages from previous phases. See field order_history to get orders from previous phases. To get a complete state of all data in this game object, consider using method Game.to_dict().

Parameters **make_copy** – Boolean. If true, a deep copy of the game state is returned, otherwise the attributes are returned directly.

Returns The internal saved state (dict) of the game

set_state(*state*, *clear_history=True*)
Sets the game from a saved internal state

Parameters

- **state** – The saved state (dict)
- **clear_history** – Boolean. If true, all game histories are cleared.

Returns Nothing

get_all_possible_orders()
Computes a list of all possible orders for all locations

Returns A dictionary with locations as keys, and their respective list of possible orders as values

4.2 diplomacy.engine.map

Map - Contains the map object which represents a map where the game can be played

class diplomacy.engine.map.**Map**(*name='standard'*, *use_cache=True*)

Bases: object

Map Class

Properties:

- **abbrev**: Contains the power abbreviation, otherwise defaults to first letter of PowerName e.g. {'ENGLISH': 'E'}
- **abuts_cache**: Contains a cache of abuts for ['A','F'] between all locations for orders ['S', 'C', '-'] e.g. {(A, PAR, -, MAR): 1, ... }
- **aliases**: Contains a dict of all the aliases (e.g. full province name to 3 char) e.g. {'EAST': 'EAS', 'STP (/SC)': 'STP/SC', 'FRENCH': 'FRANCE', 'BUDAPEST': 'BUD', 'NOR': 'NWY', ... }
- **centers**: Contains a dict of owned supply centers for each player at the beginning of the map e.g. {'RUSSIA': ['MOS', 'SEV', 'STP', 'WAR'], 'FRANCE': ['BRE', 'MAR', 'PAR'], ... }
- **convoy_paths**: Contains a list of all possible convoys paths bucketed by number of fleets format: {nb of fleets: [(START_LOC, {FLEET LOC}, {DEST LOCS})]}
- **dest_with_coasts**: Contains a dictionary of locs with all destinations (incl coasts) that can be reached e.g. {'PAR': ['BRE', 'PIC', 'BUR', ...], ... }
- **dummies**: Indicates the list of powers that are dummies e.g. ['FRANCE', 'ITALY']
- **error**: Contains a list of errors that the map generated e.g. [''DUPLICATE MAP ALIAS OR POWER: JAPAN']
- **files**: Contains a list of files that were loaded (e.g. USES keyword) e.g. ['standard.map', 'standard.politics', 'standard.geography', 'standard.military']
- **first_year**: Indicates the year where the game is starting. e.g. 1901
- **flow**: List that contains the seasons with the phases e.g. ['SPRING:MOVEMENT,RETREATS', 'FALL:MOVEMENT,RETREATS', 'WINTER:ADJUSTMENTS']
- **flow_sign**: Indicate the direction of flow (1 is positive, -1 is negative) e.g. 1
- **homes**: Contains the list of supply centers where units can be built (i.e. assigned at the beginning) e.g. {'RUSSIA': ['MOS', 'SEV', 'STP', 'WAR'], 'FRANCE': ['BRE', 'MAR', 'PAR'], ... }
- **inhabits**: List that indicates which power have a INHABITS, HOME, or HOMES line e.g. ['FRANCE']
- **keywords**: Contains a dict of keywords to parse status files and orders e.g. {'BUILDS': 'B', '>': '', 'SC': '/SC', 'REMOVING': 'D', 'WAIVED': 'V', 'ATTACK': '', ... }
- **loc_abut**: Contains a adjacency list for each province e.g. {'LVP': ['CLY', 'edi', 'IRI', 'NAO', 'WAL', 'yor'], ... }
- **loc_coasts**: Contains a mapping of all coasts for every location e.g. {'PAR': ['PAR'], 'BUL': ['BUL', 'BUL/EC', 'BUL/SC'], ... }
- **loc_name**: Dict that indicates the 3 letter name of each location e.g. {'GULF OF LYON': 'LYO', 'BREST': 'BRE', 'BUDAPEST': 'BUD', 'RUHR': 'RUH', ... }
- **loc_type**: Dict that indicates if each location is 'WATER', 'COAST', 'LAND', or 'PORT' e.g. {'MAO': 'WATER', 'SER': 'LAND', 'SYR': 'COAST', 'MOS': 'LAND', 'VEN': 'COAST', ... }
- **locs**: List of 3 letter locations (With coasts) e.g. ['ADR', 'AEG', 'ALB', 'ANK', 'APU', 'ARM', 'BAL', 'BAR', 'BEL', 'BER', ...]
- **name**: Name of the map (or full path to a custom map file) e.g. 'standard' or '/some/path/to/file.map'

- **own_word**: Dict to indicate the word used to refer to people living in each power's country e.g. {'RUSSIA': 'RUSSIAN', 'FRANCE': 'FRENCH', 'UNOWNED': 'UNOWNED', 'TURKEY': 'TURKISH', ... }
- **owns**: List that indicates which power have a OWNS or CENTERS line e.g. ['FRANCE']
- **phase**: String to indicate the beginning phase of the map e.g. 'SPRING 1901 MOVEMENT'
- **phase_abbrev**: Dict to indicate the 1 letter abbreviation for each phase e.g. {'A': 'ADJUSTMENTS', 'M': 'MOVEMENT', 'R': 'RETREATS'}
- **pow_name**: Dict to indicate the power's name e.g. {'RUSSIA': 'RUSSIA', 'FRANCE': 'FRANCE', 'TURKEY': 'TURKEY', 'GERMANY': 'GERMANY', ... }
- **powers**: Contains the list of powers (players) in the game e.g. ['AUSTRIA', 'ENGLAND', 'FRANCE', 'GERMANY', 'ITALY', 'RUSSIA', 'TURKEY']
- **root_map**: Contains the name of the original map file loaded (before the USES keyword are applied) A map that is called with MAP is the root_map. e.g. 'standard'
- **rules**: Contains a list of rules used by all variants (for display only) e.g. ['RULE_1']
- **scs**: Contains a list of all the supply centers in the game e.g. ['MOS', 'SEV', 'STP', 'WAR', 'BRE', 'MAR', 'PAR', 'BEL', 'BUL', 'DEN', 'GRE', 'HOL', 'NWY', ...]
- **seq**: [] Contains the sequence of seasons in format 'SEASON_NAME SEASON_TYPE' e.g. ['NEWYEAR', 'SPRING MOVEMENT', 'SPRING RETREATS', 'FALL MOVEMENT', 'FALL RETREATS', 'WINTER ADJUSTMENTS']
- **unclear**: Contains the alias for ambiguous places e.g. {'EAST': 'EAS'}
- **unit_names**: {} Contains a dict of the unit names e.g. {'F': 'FLEET', 'A': 'ARMY'}
- **units**: Dict that contains the current position of each unit by power e.g. {'FRANCE': ['F BRE', 'A MAR', 'A PAR'], 'RUSSIA': ['A WAR', 'A MOS', 'F SEV', 'F STP/SC'], ... }
- **validated**: Boolean to indicate if the map file has been validated e.g. 1
- **victory**: Indicates the number of supply centers to win the game (>50% required if None) e.g. 18

__init__(*name='standard'*, *use_cache=True*)
Constructor function

Parameters

- **name** – Name of the map to load (or full path to a custom map file)
- **use_cache** – Boolean flag to indicate we want a blank object that doesn't use cache

svg_path
Return path to the SVG file of this map (or None if it does not exist)

validate(*force=0*)
Validate that the configuration from a map file is correct

Parameters **force** – Indicate that we want to force a validation, even if the map is already validated

Returns Nothing

load(*file_name=None*)
Loads a map file from disk

Parameters **file_name** – Optional. A string representing the file to open. Otherwise, defaults to the map name

Returns Nothing

build_cache()
Builds a cache to speed up abuts and coasts lookup

add_homes(*power*, *homes*, *reinit*)
Add new homes (and deletes previous homes if reinit)

Parameters

- **power** – Name of power (e.g. ITALY)
- **homes** – List of homes e.g. `['BUR', '-POR', '*ITA', ... ]`
- **reinit** – Indicates that we want to strip the list of homes before adding

Returns Nothing

drop(*place*)
Drop a place

Parameters **place** – Name of place to remove

Returns Nothing

norm_power(*power*)
Normalise the name of a power (removes spaces)

Parameters **power** – Name of power to normalise

Returns Normalised power name

norm(*phrase*)
Normalise a sentence (add spaces before /, replace -+, with ' ', remove .:

Parameters **phrase** – Phrase to normalise

Returns Normalised sentences

compact(*phrase*)
Compacts a full sentence into a list of short words

Parameters **phrase** – The full sentence to compact (e.g. 'England: Fleet Western Mediterranean -> Tyrrhenian Sea. (*bounce*)')

Returns The compacted phrase in an array (e.g. ['ENGLAND', 'F', 'WES', 'TYS', 'I'])

alias(*word*)
This function is used to replace multi-words with their acronyms

Parameters **word** – The current list of words to try to shorten

Returns alias, ix - alias is the shorten list of word, ix is the ix of the next non-processed word

vet(*word*, *strict=0*)
Determines the type of every word in a compacted order phrase

0 - Undetermined, 1 - Power, 2 - Unit, 3 - Location, 4 - Coastal location 5 - Order, 6 - Move Operator `(-=_^)`, 7 - Non-move separator `(|?~)` or result `(*!?~+)`

Parameters

- **word** – The list of words to vet (e.g. `['A', 'POR', 'S', 'SPA/NC']`)
- **strict** – Boolean to indicate that we want to verify that the words actually exist. Numbers become negative if they don't exist

Returns A list of tuple (e.g. `[('A', 2), ('POR', 3), ('S', 5), ('SPA/NC', 4)]`)

rearrange(*word*)
This function is used to parse commands

Parameters **word** – The list of words to vet (e.g. ['ENGLAND', 'F', 'WES', 'TYS', 'l'])

Returns The list of words in the correct order to be processed (e.g. ['ENGLAND', 'F', 'WES', '-', 'TYS'])

area_type(*loc*)
Returns 'WATER', 'COAST', 'PORT', 'LAND', 'SHUT'

Parameters **loc** – The name of the location to query

Returns Type of the location ('WATER', 'COAST', 'PORT', 'LAND', 'SHUT')

default_coast(*word*)
Returns the coast for a fleet move order that can only be to a single coast (e.g. F GRE-BUL returns F GRE-BUL/SC)

Parameters **word** – A list of tokens (e.g. ['F', 'GRE', '-', 'BUL'])

Returns The updated list of tokens (e.g. ['F', 'GRE', '-', 'BUL/SC'])

find_coasts(*loc*)
Finds all coasts for a given location

Parameters **loc** – The name of a location (e.g. 'BUL')

Returns Returns the list of all coasts, including the location (e.g. ['BUL', 'BUL/EC', 'BUL/SC']

abuts(*unit_type*, *unit_loc*, *order_type*, *other_loc*)
Determines if a order for unit_type from unit_loc to other_loc is adjacent.

Note: This method uses the precomputed cache

Parameters

- **unit_type** – The type of unit ('A' or 'F')
- **unit_loc** – The location of the unit ('BUR', 'BUL/EC')
- **order_type** – The type of order ('S' for Support, 'C' for Convoy', '-' for move)
- **other_loc** – The location of the other unit

Returns 1 if the locations are adjacent for the move, 0 otherwise

is_valid_unit(*unit*, *no_coast_ok=0*, *shut_ok=0*)
Determines if a unit and location combination is valid (e.g. 'A BUR') is valid

Parameters

- **unit** – The name of the unit with its location (e.g. F SPA/SC)
- **no_coast_ok** – Indicates if a coastal location with no coast (e.g. SPA vs SPA/SC) is acceptable
- **shut_ok** – Indicates if a impassable country (e.g. Switzerland) is OK

Returns A boolean to indicate if the unit/location combination is valid

abut_list(*site*, *incl_no_coast=False*)
Returns the adjacency list for the site

Parameters

- **site** – The province we want the adjacency list for
- **incl_no_coast** – Boolean flag that indicates to also include province without coast if it has coasts e.g. will return ['BUL/SC', 'BUL/EC'] if False, and ['bul', 'BUL/SC', 'BUL/EC'] if True

Returns A list of adjacent provinces

Note: abuts are returned in **mixed cases**

- An adjacency that is lowercase (e.g. 'bur') can only be used by an army
- An adjacency that starts with a capital letter (e.g. 'Bal') can only be used by a fleet
- An adjacency that is uppercase can be used by both an army and a fleet

find_next_phase(*phase*, *phase_type=None*, *skip=0*)
Returns the long name of the phase coming immediately after the phase

Parameters

- **phase** – The long name of the current phase (e.g. SPRING 1905 RETREATS)
- **phase_type** – The type of phase we are looking for (e.g. 'M' for Movement, 'R' for Retreats, 'A' for Adjust.)
- **skip** – The number of match to skip (e.g. 1 to find not the next phase, but the one after)

Returns The long name of the next phase (e.g. FALL 1905 MOVEMENT)

find_previous_phase(*phase*, *phase_type=None*, *skip=0*)
Returns the long name of the phase coming immediately prior the phase

Parameters

- **phase** – The long name of the current phase (e.g. SPRING 1905 RETREATS)
- **phase_type** – The type of phase we are looking for (e.g. 'M' for Movement, 'R' for Retreats, 'A' for Adjust.)
- **skip** – The number of match to skip (e.g. 1 to find not the next phase, but the one after)

Returns The long name of the previous phase (e.g. SPRING 1905 MOVEMENT)

compare_phases(*phase1*, *phase2*)
Compare 2 phases (Strings) and return 1, -1, or 0 to indicate which phase is larger

Parameters

- **phase1** – The first phase (e.g. S1901M, FORMING, COMPLETED)
- **phase2** – The second phase (e.g. S1901M, FORMING, COMPLETED)

Returns 1 if phase1 > phase2, -1 if phase2 > phase1 otherwise 0 if they are equal

static phase_abbr(*phase*, *default='?????'*)
Constructs a 5 character representation (S1901M) from a phase (SPRING 1901 MOVEMENT)

Parameters

- **phase** – The full phase (e.g. SPRING 1901 MOVEMENT)
- **default** – The default value to return in case conversion fails

Returns A 5 character representation of the phase

phase_long(*phase_abbr*, *default='?????'*)
Constructs a full sentence of a phase from a 5 character abbreviation

Parameters

- **phase_abbr** – 5 character abbrev. (e.g. S1901M)
- **default** – The default value to return in case conversion fails

Returns A full phase description (e.g. SPRING 1901 MOVEMENT)

4.3 diplomacy.engine.message

Game message. Represent a message exchanged inside a game.

Possible messages exchanges:

- power 1 -> power 2
- power -> all game
- system -> power
- system -> all game
- system -> observers
- system -> omniscient observers

Sender *system* is identified with constant SYSTEM defined below.

Recipients *all game*, *observers* and *omniscient observers* are identified respectively with constants GLOBAL, OBSERVER and OMNISCIENT defined below.

Consider using Game methods to generate appropriate messages instead of this class directly:

- Game.new_power_message() to send a message from a power to another.
- Game.new_global_message() to send a message from a power to all game.
- ServerGame.new_system_message() to send a server system message. Use constant names defined below to specify recipient for system message when it's not a power name (GLOBAL, OBSERVER or OMNISCIENT).

class diplomacy.engine.message.**Message**(***kwargs*)

Bases: diplomacy.utils.jsonable.Jsonable

Message class.

Properties:

- **sender**: message sender name: either SYSTEM or a power name.
- **recipient**: message recipient name: either GLOBAL, OBSERVER, OMNISCIENT or a power name.
- **time_sent**: message timestamp in microseconds.
- **phase**: short name of game phase when message is sent.
- **message**: message body.

Note about timestamp management:

We assume a message has an unique timestamp inside one game. To respect this rule, the server is the only one responsible for generating message timestamps. This allow to generate timestamp or only 1 same machine (server) instead of managing timestamps from many user machines, to prevent timestamp inconsistency when messages are stored on server. Therefore, message timestamp is the time when server stores the message, not the time when message was sent by any client.

is_global()
Return True if this message is global.

for_observer()
Return True if this message is sent to observers.

4.4 diplomacy.engine.power

Power

- Contains the power object representing a power in the game

class diplomacy.engine.power.**Power**(*game=None*, *name=None*, ***kwargs*)
Bases: diplomacy.utils.jsonable.Jsonable

Power Class

Properties:

- **abbrev** - Contains the abbrev of the power (i.e. the first letter of the power name) (e.g. 'F' for FRANCE)
- **adjust** - List of pending adjustment orders (e.g. ['A PAR B', 'A PAR R MAR', 'A MAR D', 'WAIVE'])
- **centers** - Contains the list of supply centers currently controlled by the power ['MOS', 'SEV', 'STP']
- **civil_disorder** - Bool flag to indicate that the power has been put in CIVIL_DISORDER (e.g. True or False)
- **controller** - Sorted dictionary mapping timestamp to controller (either dummy or a user ID) who takes control of power at this timestamp.
- **game** - Contains a reference to the game object
- **goner** - Boolean to indicate that this power doesn't control any SCs any more (e.g. True or False)
- **homes** - Contains a list of homes supply centers (where you can build) e.g. ['PAR', 'MAR', ...] or None if empty
- **influence** - Contains a list of locations influenced by this power Note: To influence a location, the power must have visited it last. e.g ['PAR', 'MAR', ...]
- **name** - Contains the name of the power (e.g. 'FRANCE')
- **orders** - Contains a dictionary of units and their orders. For NO_CHECK games, unit is 'ORDER 1', 'ORDER 2', ...
  - e.g. {'A PAR': '- MAR' } or {'ORDER 1': 'A PAR - MAR', 'ORDER 2': '...', ... }
  - Can also be {'REORDER 1': 'A PAR - MAR', 'INVALID 1': 'A PAR - MAR', ... } after validation
- **retreats** - Contains the list of units that need to retreat with their possible retreat locations (e.g. {'A PAR': ['MAR', 'BER']})
- **role** - Power type (observer, omniscient, player or server power). Either the power name (for a player power) or a value in diplomacy.utils.strings.ALL_ROLE_TYPES
- **tokens** - Only for server power: set of tokens of current power controlled (if not None).
- **units** - Contains the list of units (e.g. ['A PAR', 'A MAR', ...]
- **vote** - Only for omniscient, player and server power: power vote ('yes', 'no' or 'neutral').

__init__(*game=None*, *name=None*, ***kwargs*)
Constructor

reinit(*include_flags=6*)
Performs a reinitialization of some of the parameters

Parameters **include_flags** – Bit mask to indicate which params to reset (bit 1 = orders, 2 = persistent, 4 = transient)

Returns None

static compare(*power_1*, *power_2*)
Comparator object - Compares two Power objects

Parameters

- **power_1** – The first Power object to compare
- **power_2** – The second Power object to compare

Returns 1 if self is greater, -1 if other is greater, 0 if they are equal

initialize(*game*)
Initializes a game and resets home, centers and units

Parameters **game** (*diplomacy.Game*) – The game to use for initialization

merge(*other_power*)
Transfer all units, centers, and homes of the other_power to this power

Parameters **other_power** – The other power (will be empty after the merge)

clear_units()
Removes all units from the map

clear_centers()
Removes ownership of all supply centers

is_dummy()
Indicates if the power is a dummy

Returns Boolean flag to indicate if the power is a dummy

is_eliminated()
Returns a flag to show if player is eliminated

Returns If the current power is eliminated

clear_orders()
Clears the power's orders

moves_submitted()
Returns a boolean to indicate if moves has been submitted

Returns 1 if not in Movement phase, or orders submitted, or no more units lefts

is_observer_power()
(Network Method) Return True if this power is an observer power.

is_omniscient_power()
(Network Method) Return True if this power is an omniscient power.

is_player_power()
(Network Method) Return True if this power is a player power.

is_server_power()
(Network Method) Return True if this power is a server power.

is_controlled()
(Network Method) Return True if this power is controlled.

does_not_wait()
(Network Method) Return True if this power does not wait (ie. if we could already process orders of this power).

update_controller(*username*, *timestamp*)
(Network Method) Update controller with given username and timestamp.

set_controlled(*username*)
(Network Method) Control power with given username. Username may be None (meaning no controller).

get_controller()
(Network Method) Return current power controller name ('dummy' if power is not controlled).

get_controller_timestamp()
(Network Method) Return timestamp when current controller took control of this power.

is_controlled_by(*username*)
(Network Method) Return True if this power is controlled by given username.

has_token(*token*)
(Server Method) Return True if this power has given token.

add_token(*token*)
(Server Method) Add given token to this power.

remove_tokens(*tokens*)
(Server Method) Remove sequence of tokens from this power.

4.5 diplomacy.engine.renderer

Renderer

- Contains the renderer object which is responsible for rendering a game state to svg

class diplomacy.engine.renderer.**Renderer**(*game*, *svg_path=None*)
Bases: object

Renderer object responsible for rendering a game state to svg

__init__(*game*, *svg_path=None*)
Constructor

Parameters

- **game** (*diplomacy.Game*) – The instantiated game object to render
- **svg_path** (*str, optional*) – Optional. Can be set to the full path of a custom SVG to use for rendering the map.

render(*incl_orders=True*, *incl_abbrev=False*, *output_format='svg'*, *output_path=None*)
Renders the current game and returns the XML representation

Parameters

- **incl_orders** (*bool, optional*) – Optional. Flag to indicate we also want to render orders.
- **incl_abbrev** (*bool, optional*) – Optional. Flag to indicate we also want to display the provinces abbreviations.

- **output_format** (*str, optional*) – The desired output format. Valid values are: 'svg'
- **output_path** (*str | None, optional*) – Optional. The full path where to save the rendering on disk.

Returns The rendered image in the specified format.

CHAPTER 5

diplomacy.integration

5.1 integration.webdiplomacy_net.api

Contains an API class to send requests to webdiplomacy.net

class diplomacy.integration.webdiplomacy_net.api.**API**(*api_key*, *connect_timeout=30*, *request_timeout=60*)

Bases: diplomacy.integration.base_api.BaseAPI

API to interact with webdiplomacy.net

list_games_with_players_in_cd()

Lists the game on the standard map where a player is in CD (civil disorder) and the bots needs to submit orders

Returns List of GameIdCountryId tuples [(game_id, country_id), (game_id, country_id)]

list_games_with_missing_orders()

Lists of the game on the standard where the user has not submitted orders yet.

Returns List of GameIdCountryId tuples [(game_id, country_id), (game_id, country_id)]

get_game_and_power(*game_id*, *country_id*, *max_phases=None*)

Returns the game and the power we are playing

Parameters

- **game_id** (*int*) – The id of the game object (integer)
- **country_id** (*int*) – The id of the country for which we want the game state (integer)
- **max_phases** (*int | None, optional*) – Optional. If set, improve speed by generating game only using the last 'x' phases.

Returns

A tuple consisting of

1. The diplomacy.Game object from the game state or None if an error occurred

2. The power name (e.g. 'FRANCE') referred to by country_id

set_orders(*game*, *power_name*, *orders*, *wait=None*)
Submits orders back to the server

Parameters

- **game** (*diplomacy.Game*) – A *diplomacy.engine.game.Game* object representing the current state of the game
- **power_name** (*str*) – The name of the power submitting the orders (e.g. 'FRANCE')
- **orders** (*List[str]*) – A list of strings representing the orders (e.g. ['A PAR H', 'F BRE - MAO'])
- **wait** (*bool | None, optional*) – Optional. If True, sets ready=False, if False sets ready=True.

Returns True for success, False for failure

CHAPTER 6

diplomacy.utils

6.1 diplomacy.utils.errors

Error - Contains the error messages and code used by the engine

class diplomacy.utils.errors.**Error**(*code*, *message=None*)
Bases: diplomacy.utils.common.StringableCode

Represents an error

class diplomacy.utils.errors.**MapError**(*code*, *message*)
Bases: *diplomacy.utils.errors.Error*

Represents a map error

__init__(*code*, *message*)
Build a MapError

Parameters

- **code** – int code of the error
- **message** – human readable string message associated to the error

class diplomacy.utils.errors.**GameError**(*code*, *message*)
Bases: *diplomacy.utils.errors.Error*

Represents a game error

__init__(*code*, *message*)
Build a GameError

Parameters

- **code** – int code of the error
- **message** – human readable string message associated to the error

class diplomacy.utils.errors.**StdError**(*code*, *message*)
Bases: *diplomacy.utils.errors.Error*

Represents a standard error

__init__(*code*, *message*)
Build a StdError

Parameters

- **code** – int code of the error
- **message** – human readable string message associated to the error

6.2 diplomacy.utils.exceptions

Exceptions used in diplomacy network code.

exception diplomacy.utils.exceptions.**DiplomacyException**(*message=''*)
Bases: Exception

Diplomacy network code exception.

exception diplomacy.utils.exceptions.**AlreadyScheduledException**(*message=''*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Cannot add a data already scheduled.

exception diplomacy.utils.exceptions.**CommonKeyException**(*key*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Common key error.

exception diplomacy.utils.exceptions.**KeyException**(*key*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Key error.

exception diplomacy.utils.exceptions.**LengthException**(*expected_length*, *given_length*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Length error.

exception diplomacy.utils.exceptions.**NaturalIntegerException**(*integer_name=''*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Expected a positive integer (int >= 0).

exception diplomacy.utils.exceptions.**NaturalIntegerNotNullException**(*integer_name=''*)
Bases: *diplomacy.utils.exceptions.NaturalIntegerException*

Expected a strictly positive integer (int > 0).

exception diplomacy.utils.exceptions.**RandomPowerException**(*nb_powers*, *nb_available_powers*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

No enough playable powers to select random powers.

exception diplomacy.utils.exceptions.**TypeException**(*expected_type*, *given_type*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Type error.

exception diplomacy.utils.exceptions.**ValueException**(*expected_values*, *given_value*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Value error.

exception diplomacy.utils.exceptions.**NotificationException**(*message=''*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Unknown notification.

exception diplomacy.utils.exceptions.**ResponseException**(*message=''*)
Bases: *diplomacy.utils.exceptions.DiplomacyException*

Unknown response.

exception diplomacy.utils.exceptions.**RequestException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Unknown request.

exception diplomacy.utils.exceptions.**AdminTokenException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid token for admin operations.

exception diplomacy.utils.exceptions.**DaidePortException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Daide server not started for the game

exception diplomacy.utils.exceptions.**GameCanceledException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Game was cancelled.

exception diplomacy.utils.exceptions.**GameCreationException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Cannot create more games on that server.

exception diplomacy.utils.exceptions.**GameFinishedException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

This game is finished.

exception diplomacy.utils.exceptions.**GameIdException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid game ID.

exception diplomacy.utils.exceptions.**GameJoinRoleException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

A token can have only one role inside a game: player, observer or omniscient.

exception diplomacy.utils.exceptions.**GameRoleException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Game role does not accepts this action.

exception diplomacy.utils.exceptions.**GameMasterTokenException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid token for master operations.

exception diplomacy.utils.exceptions.**GameNotPlayingException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Game not playing.

exception diplomacy.utils.exceptions.**GameObserverException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Disallowed observation for non-master users.

exception diplomacy.utils.exceptions.**GamePhaseException**(*expected=None*, *given=None*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Data does not match current game phase.

exception diplomacy.utils.exceptions.**GamePlayerException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid player.

exception diplomacy.utils.exceptions.**GameRegistrationPasswordException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid game registration password.

exception diplomacy.utils.exceptions.**GameSolitaireException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

A solitaire game does not accepts players.

exception diplomacy.utils.exceptions.**GameTokenException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid token for this game.

exception diplomacy.utils.exceptions.**MapIdException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid map ID.

exception diplomacy.utils.exceptions.**MapPowerException**(*power_name*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid map power.

exception diplomacy.utils.exceptions.**FolderException**(*folder_path*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Given folder not available in server.

exception diplomacy.utils.exceptions.**ServerRegistrationException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Registration currently not allowed on this server.

exception diplomacy.utils.exceptions.**TokenException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid token.

exception diplomacy.utils.exceptions.**UserException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Invalid user.

exception diplomacy.utils.exceptions.**PasswordException**(*message=''*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Password must not be empty.

exception diplomacy.utils.exceptions.**ServerDirException**(*server_dir*)
Bases: *diplomacy.utils.exceptions.ResponseException*

Error with working folder.

6.3 diplomacy.utils.export

Exporter - Responsible for exporting games in a standardized format to disk

diplomacy.utils.export.**to_saved_game_format**(*game*, *output_path=None*, *output_mode='a'*)
Converts a game to a standardized JSON format

Parameters

- **game** (diplomacy.engine.game.Game) – game to convert.
- **output_path** (*str | None, optional*) – Optional path to file. If set, the json.dumps() of the saved_game is written to that file.
- **output_mode** (*str, optional*) – Optional. The mode to use to write to the output_path (if provided). Defaults to 'a'

Returns A game in the standard format used to saved game, that can be converted to JSON for serialization

Return type Dict

diplomacy.utils.export.**from_saved_game_format**(*saved_game*)
Rebuilds a *diplomacy.engine.game.Game* object from the saved game (python `Dict`) saved_game is the dictionary. It can be built by calling json.loads(json_line).

Parameters **saved_game** (*Dict*) – The saved game exported from *to_saved_game_format()*

Return type *diplomacy.engine.game.Game*

Returns The game object restored from the saved game

diplomacy.utils.export.**load_saved_games_from_disk**(*input_path*, *on_error='raise'*)
Rebuids multiple *diplomacy.engine.game.Game* from each line in a .jsonl file

Parameters

- **input_path** (*str*) – The path to the input file. Expected content is one saved_game json per line.
- **on_error** – Optional. What to do if a game conversion fails. Either 'raise', 'warn', 'ignore'

Return type List[diplomacy.Game]

Returns A list of *diplomacy.engine.game.Game* objects.

diplomacy.utils.export.**is_valid_saved_game**(*saved_game*)
Checks if the saved game is valid. This is an expensive operation because it replays the game.

Parameters **saved_game** – The saved game (from to_saved_game_format)

Returns A boolean that indicates if the game is valid

6.4 diplomacy.utils.order_results

Results

- Contains the results labels and code used by the engine

class diplomacy.utils.order_results.**OrderResult**(*code*, *message*)
Bases: diplomacy.utils.common.StringableCode

Represents an order result

__init__(*code*, *message*)
Build a Order Result

Parameters

- **code** – int code of the order result
- **message** – human readable string message associated to the order result

diplomacy.utils.order_results.**OK = 0:**
Order result OK, printed as ''

diplomacy.utils.order_results.**NO_CONVOY = 10001:no convoy**
Order result NO_CONVOY, printed as 'no convoy'

diplomacy.utils.order_results.**BOUNCE = 10002:bounce**
Order result BOUNCE, printed as 'bounce'

diplomacy.utils.order_results.**VOID = 10003:void**
Order result VOID, printed as 'void'

diplomacy.utils.order_results.**CUT = 10004:cut**
Order result CUT, printed as 'cut'

diplomacy.utils.order_results.**DISLODGED = 10005:dislodged**
Order result DISLODGED, printed as 'dislodged'

diplomacy.utils.order_results.**DISRUPTED = 10006:disrupted**
Order result DISRUPTED, printed as 'disrupted'

diplomacy.utils.order_results.**DISBAND = 10007:disband**
Order result DISBAND, printed as 'disband'

diplomacy.utils.order_results.**MAYBE = 10008:maybe**
Order result MAYBE, printed as 'maybe'

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y