TextWorld Documentation

Marc-Alexandre Côté, Ákos Kádár, Xingdi Yuan, Ben Kybartas, Ta

Sep 17, 2022

CONTENTS

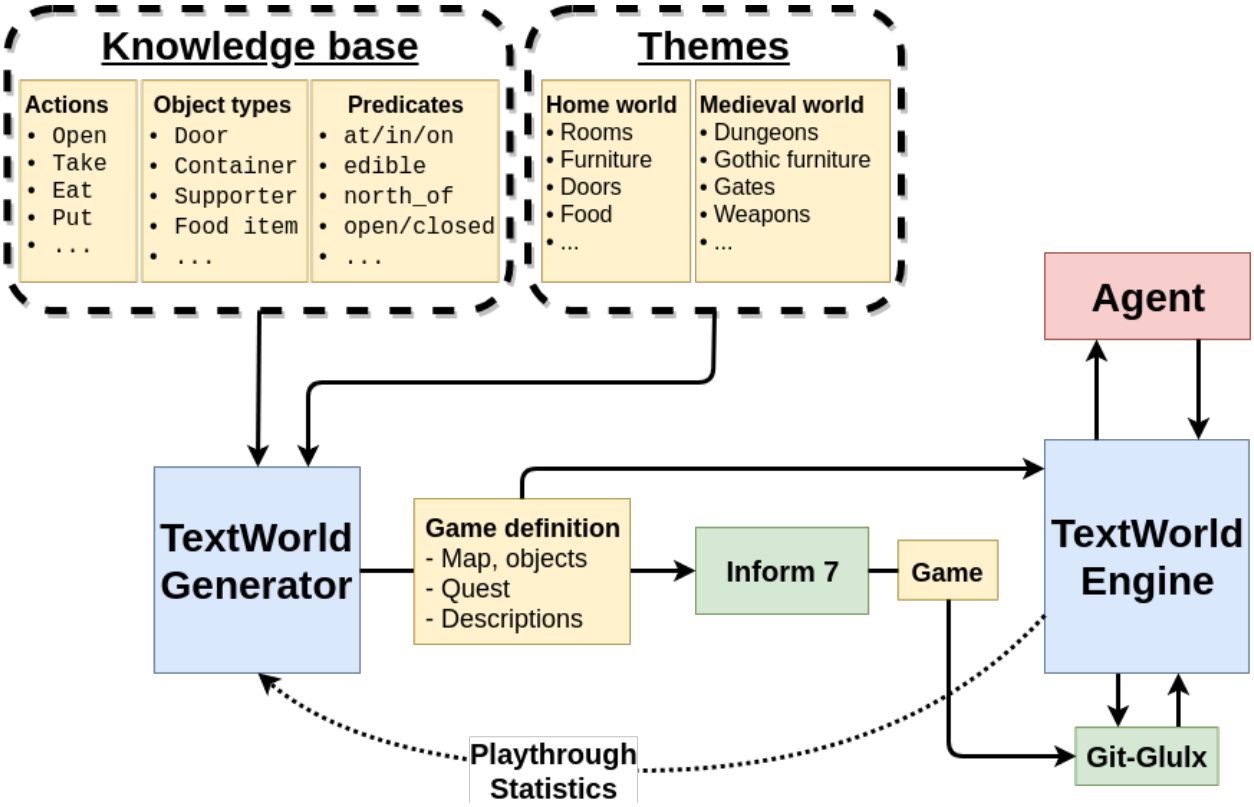
TextWorld is a text-based learning environment for Reinforcement Learning agent.

CHAPTER

ONE

WHAT IS TEXTWORLD?

TextWorld is a sandbox learning environment for training and testing reinforcement learning (RL) agents on text-based games. It enables generating games from a game distribution parameterized by the map size, the number of objects, quest length and complexity, richness of text descriptions, and more. Then, one can sample game from that distribution. TextWorld can also be used to play existing text-based games.



CHAPTER

TWO

KNOWN ISSUES

2.1 Inform 7

Inform 7 command line tools don't support Windows Linux Subsystem (a.k.a Bash on Ubuntu on Windows).

CHAPTER

THREE

TW-PLAY

Play a TextWorld game (.z8 or .ulx).

```
usage: tw-play [-h] [--mode MODE] [--max-steps STEPS] [--viewer [PORT]]
               [--hint] [-v] [-vv]
               game
```

3.1 Positional Arguments

game

3.2 Named Arguments

--mode Possible choices: random, human, random-cmd, walkthrough

Select an agent to play the game: ['random', 'human', 'random-cmd', 'walkthrough']. Default: "human".

Default: "human"

--max-steps Limit maximum number of steps.

Default: 0

--viewer Start web viewer.

--hint Display the oracle trajectory leading to winning the game.

Default: False

-v, --verbose Verbose mode.

Default: False

-vv, --very-verbose Print debug information.

Default: False

CHAPTER

FOUR

TW-MAKE

```
usage: tw-make [-h] [--third-party PATH]
               {custom,tw-coin_collector,tw-treasure_hunter,tw-simple,tw-cooking}
               ...
```

4.1 Named Arguments

--third-party Load an external python file. Useful to register custom challenges on-the-fly.

4.2 Types of game to create

subcommand Possible choices: custom, tw-coin_collector, tw-treasure_hunter, tw-simple, tw-cooking

4.3 Sub-commands:

4.3.1 custom

Make a custom game.

```
tw-make custom [-h] [--world-size SIZE] [--nb-objects NB] [--theme THEME]
               [--include-adj] [--blend-descriptions]
               [--ambiguous-instructions] [--only-last-action]
               [--blend-instructions] [--entity-numbering]
               [--nb-parallel-quests NB_PARALLEL_QUESTS]
               [--quest-length LENGTH] [--quest-breadth BREADTH]
               [--quest-min-length LENGTH] [--quest-max-length LENGTH]
               [--quest-min-breadth BREADTH] [--quest-max-breadth BREADTH]
               [--quest-min-depth DEPTH] [--quest-max-depth DEPTH]
               [--output PATH] [--seed SEED] [--format {ulx,z8}] [--overview]
               [--save-overview] [-f] [--silent | -v]
```

Custom game settings

--world-size Nb. of rooms in the world.

Default: 5

--nb-objects Minimum nb. of objects in the world.

Default: 10

Grammar settings

--theme Theme to use for generating the text. Default: "house"

Default: "house"

--include-adj Turn on adjectives.

Default: False

--blend-descriptions Blend descriptions across consecutive sentences.

Default: False

--ambiguous-instructions Refer to an object using its type (e.g. red container vs. red chest).

Default: False

--only-last-action Intruction only describes the last action of quest.

Default: False

--blend-instructions Blend instructions across consecutive actions.

Default: False

--entity-numbering Append a number after an entity name if there is not enough variation for it (e.g. 'red apple 2').

Default: False

Quest settings

--nb-parallel-quests Nb. of parallel quests the game will have. Default: 1.

Default: 1

--quest-length Nb. of actions the quest requires to be completed. It is a shorthand for '–quest-min-length N –quest-max-length N –quest-max-depth N'.

--quest-breadth Nb. of subquests the quests will have. It is a shorthand for '–quest-min-breadth N –quest-max-breadth N'.

Quest settings (advanced)

--quest-min-length Minimum nb. of actions the quest requires to be completed. This setting is ignored if –quest-length is provided. Default: 1.

Default: 1

--quest-max-length Maximum nb. of actions the quest requires to be completed. This setting is ignored if –quest-length is provided. Default: 5.

Default: 5

--quest-min-breadth Minimum nb. of subquests the quests can have. This setting is ignored if –quest-breadth is provided. Default: 1.

Default: 1

--quest-max-breadth Maxmimum nb. of subquests the quests can have. This setting is ignored if –quest-breadth is provided. Default: 5.

Default: 5

--quest-min-depth Minimum nb. of actions the subquests can have. Default: 1.

Default: 1

--quest-max-depth Maximum nb. of actions the subquests can have. This setting is ignored if –quest-length is provided. Default: 5.

Default: 5

General settings

--output Path where to save the generated game. If it points to a folder, the game's UUID will be used as the filename.

Default: "./tw_games/"

--seed

--format Possible choices: ulx, z8

Which format to use when compiling the game. Default: "z8"

Default: "z8"

--overview Display an overview of the generated game.

Default: False

--save-overview Save the overview image of the generated game alongside the game as a PNG file.

Default: False

-f, --force Default: False

--silent Default: False

-v, --verbose Default: False

4.3.2 tw-coin_collector

Generate a Coin Collector game

```
tw-make tw-coin_collector [-h] --level LEVEL [--output PATH] [--seed SEED]
                          [--format {ulx,z8}] [--overview] [--save-overview]
                          [-f] [--silent | -v]
```

Coin Collector game settings

--level The difficulty level. Must be between 1 and 300 (included).

General settings

--output Path where to save the generated game. If it points to a folder, the game's UUID will be used as the filename.

Default: "./tw_games/"

--seed

--format Possible choices: ulx, z8

Which format to use when compiling the game. Default: "z8"

Default: "z8"

--overview Display an overview of the generated game.

Default: False

--save-overview Save the overview image of the generated game alongside the game as a PNG file.

Default: False

-f, --force Default: False

--silent Default: False

-v, --verbose Default: False

4.3.3 tw-treasure_hunter

Generate a Treasure Hunter game

```
tw-make tw-treasure_hunter [-h] --level LEVEL [--output PATH] [--seed SEED]
                           [--format {ulx,z8}] [--overview] [--save-overview]
                           [-f] [--silent | -v]
```

Treasure Hunter game settings

--level The difficulty level. Must be between 1 and 30 (included).

General settings

--output Path where to save the generated game. If it points to a folder, the game's UUID will be used as the filename.

Default: "./tw_games/"

--seed

--format Possible choices: ulx, z8

Which format to use when compiling the game. Default: "z8"

Default: "z8"

--overview Display an overview of the generated game.

Default: False

--save-overview Save the overview image of the generated game alongside the game as a PNG file.

Default: False

-f, --force Default: False

--silent Default: False

-v, --verbose Default: False

4.3.4 tw-simple

Generate simple challenge game

```
tw-make tw-simple [-h] --rewards {dense,balanced,sparse} --goal
                  {detailed,brief,none} [--test] [--output PATH] [--seed SEED]
                  [--format {ulx,z8}] [--overview] [--save-overview] [-f]
                  [--silent | -v]
```

Simple game settings

--rewards Possible choices: dense, balanced, sparse

The reward frequency: dense, balanced, or sparse.

--goal Possible choices: detailed, brief, none

The description of the game's objective shown at the beginning of the game: detailed, bried, or none

--test Whether this game should be drawn from the test distributions of games.

Default: False

General settings

--output Path where to save the generated game. If it points to a folder, the game's UUID will be used as the filename.

Default: "./tw_games/"

--seed

--format Possible choices: ulx, z8

Which format to use when compiling the game. Default: "z8"

Default: "z8"

--overview Display an overview of the generated game.

Default: False

--save-overview Save the overview image of the generated game alongside the game as a PNG file.

Default: False

-f, --force Default: False

--silent Default: False

-v, --verbose Default: False

4.3.5 tw-cooking

Generate cooking games similar to those used for the First TextWorld Problem (FTWP) competition (https://aka.ms/ftwp).

```
tw-make tw-cooking [-h] [--recipe INT] [--take INT] [--go {1,6,9,12}] [--open]
                   [--cook] [--cut] [--drop] [--recipe-seed INT]
                   [--split {train,valid,test}] [--output PATH] [--seed SEED]
                   [--format {ulx,z8}] [--overview] [--save-overview] [-f]
                   [--silent | -v]
```

The Cooking Game settings

--recipe Number of ingredients in the recipe. Default: 1

Default: 1

--take Number of ingredients to find. It must be less or equal to the value of `--recipe`. Default: 0

Default: 0

--go Possible choices: 1, 6, 9, 12

Number of locations in the game (1, 6, 9, or 12). Default: 1

Default: 1

--open Whether containers/doors need to be opened.

Default: False

--cook Whether some ingredients need to be cooked.

Default: False

--cut Whether some ingredients need to be cut.

Default: False

--drop Whether the player's inventory has limited capacity.

Default: False

--recipe-seed Random seed used for generating the recipe. Default: 0

Default: 0

--split Possible choices: train, valid, test

Specify the game distribution to use. Food items (adj-noun pairs) are split in three subsets. Also, the way the training food items can be prepared is further divided in three subsets.

- train: training food and their corresponding training preparations
- valid: valid food + training food but with unseen valid preparations
- test: test food + training food but with unseen test preparations

Default: game is drawn from the joint distribution over train, valid, and test.

General settings

--output Path where to save the generated game. If it points to a folder, the game's UUID will be used as the filename.

Default: "./tw_games/"

--seed

--format Possible choices: ulx, z8

Which format to use when compiling the game. Default: "z8"

Default: "z8"

--overview Display an overview of the generated game.

Default: False

--save-overview Save the overview image of the generated game alongside the game as a PNG file.

Default: False

-f, --force Default: False

--silent Default: False

-v, --verbose Default: False

CHAPTER

FIVE

TW-VIEW

Display the graph representation of a game's initial state.

```
usage: tw-view [-h] [-v] game
```

5.1 Positional Arguments

game JSON file containing infos about the game.

5.2 Named Arguments

-v, --verbose Verbose mode.

Default: False

CHAPTER

SIX

TW-EXTRACT

Extract information from of a list of TextWorld games.

```
usage: tw-extract [-h] [-f] [--merge] [-q | -v]
                  {vocab,entities,walkthroughs,commands} ...
```

6.1 Positional Arguments

subcommand Possible choices: vocab, entities, walkthroughs, commands

Type of information to extract.

6.2 Named Arguments

-q, --quiet Default: False

-v, --verbose Default: False

6.3 General settings

-f, --force Default: False

--merge Merge extracted information to existing output file.

Default: False

6.4 Sub-commands:

6.4.1 vocab

Extract vocabulary.

```
tw-extract vocab [-h] [-f] [--merge] [-q | -v] [--output OUTPUT]
                 [--theme THEME]
                 [game [game ...]]
```

Positional Arguments

game List of TextWorld games (.ulx|.z8|.json).

Named Arguments

-q, --quiet Default: False

-v, --verbose Default: False

--output Output file containing all words (.txt). Default: "vocab.txt"

Default: "vocab.txt"

--theme Provide a text grammar theme from which to extract words.

General settings

-f, --force Default: False

--merge Merge extracted information to existing output file.

Default: False

6.4.2 entities

Extract entity names.

```
tw-extract entities [-h] [-f] [--merge] [-q | -v] [--output OUTPUT]
                    game [game ...]
```

Positional Arguments

game List of TextWorld games (.ulx|.z8|.json).

Named Arguments

-q, --quiet Default: False

-v, --verbose Default: False

--output Output file containing all entity names (.txt). Default: "entities.txt"

Default: "entities.txt"

General settings

-f, --force Default: False

--merge Merge extracted information to existing output file.
Default: False

6.4.3 walkthroughs

Extract walkthroughs.

```
tw-extract walkthroughs [-h] [-f] [--merge] [-q | -v] [--output OUTPUT]
                        game [game ...]
```

Positional Arguments

game List of TextWorld games (.ulx|.json).

Named Arguments

-q, --quiet Default: False

-v, --verbose Default: False

--output Output file containing all walkthroughs (.txt). Default: "walkthroughs.txt"
Default: "walkthroughs.txt"

General settings

-f, --force Default: False

--merge Merge extracted information to existing output file.
Default: False

6.4.4 commands

Extract all possible commands.

```
tw-extract commands [-h] [-f] [--merge] [-q | -v] [--output OUTPUT]
                    game [game ...]
```

Positional Arguments

game List of TextWorld games (.ulx|.json).

Named Arguments

-q, --quiet Default: False

-v, --verbose Default: False

--output Output file containing all commands (.txt). Default: “commands.txt”

Default: “commands.txt”

General settings

-f, --force Default: False

--merge Merge extracted information to existing output file.

Default: False

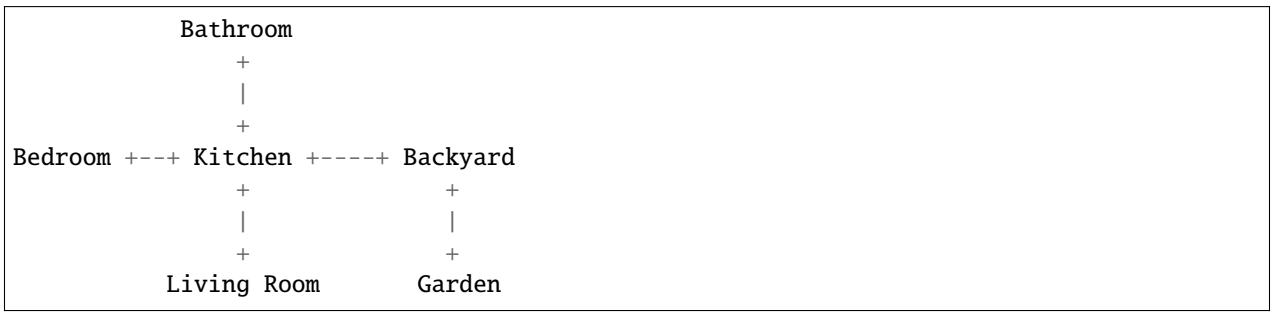
CHAPTER

SEVEN

A SIMPLE GAME

This simple game takes place in a typical house and consists in finding the right food item and cooking it.

Here's the map of the house.



7.1 Usage

```
usage: tw-make tw-simple [-h] --rewards {dense,balanced,sparse} --goal
                         {detailed,brief,none} [--test]
```

7.1.1 Simple game settings

--rewards Possible choices: dense, balanced, sparse

The reward frequency: dense, balanced, or sparse.

--goal Possible choices: detailed, brief, none

The description of the game's objective shown at the beginning of the game: detailed, bried, or none

--test Whether this game should be drawn from the test distributions of games.

Default: False

CHAPTER

EIGHT

COIN COLLECTOR

In this type of game, the world consists in a chain of `quest_length` rooms with potentially distractors rooms (i.e. leading to a dead end). The agent stats on one end and has to collect a "coin" object which is placed at the other end. There is no other objects present in the world other than the coin to collect.

8.1 Usage

```
usage: tw-make tw-coin_collector [-h] --level LEVEL
```

8.1.1 Coin Collector game settings

--level The difficulty level. Must be between 1 and 300 (included).

CHAPTER

NINE

THE COOKING GAME

This type of game was used for the competition *First TextWorld Problems*[1]. The overall objective of the game is to locate the kitchen, read the cookbook, fetch the recipe's ingredients, process them accordingly, prepare the meal, and eat it. To control the game's difficulty, one can specify the amount of skills that are involved to solve it (see skills section below).

References

9.1 Usage

```
usage: tw-make tw-cooking [-h] [--recipe INT] [--take INT] [--go {1,6,9,12}]
                          [--open] [--cook] [--cut] [--drop]
                          [--recipe-seed INT] [--split {train,valid,test}]
```

9.1.1 The Cooking Game settings

--recipe Number of ingredients in the recipe. Default: 1

Default: 1

--take Number of ingredients to find. It must be less or equal to the value of `--recipe`. Default: 0

Default: 0

--go Possible choices: 1, 6, 9, 12

Number of locations in the game (1, 6, 9, or 12). Default: 1

Default: 1

--open Whether containers/doors need to be opened.

Default: False

--cook Whether some ingredients need to be cooked.

Default: False

--cut Whether some ingredients need to be cut.

Default: False

[1] https://aka.ms/ftwp

--drop Whether the player's inventory has limited capacity.

Default: False

--recipe-seed Random seed used for generating the recipe. Default: 0

Default: 0

--split Possible choices: train, valid, test

Specify the game distribution to use. Food items (adj-noun pairs) are split in three subsets. Also, the way the training food items can be prepared is further divided in three subsets.

- train: training food and their corresponding training preparations
- valid: valid food + training food but with unseen valid preparations
- test: test food + training food but with unseen test preparations

Default: game is drawn from the joint distribution over train, valid, and test.

CHAPTER

TEN

TREASURE HUNTER

In this type of game, the agent spawns in a randomly generated maze and must find a specific object which is mentioned in the objective displayed when game starts. This is a text version of the task proposed in [Parisotto2017].

References

10.1 Usage

```
usage: tw-make tw-treasure_hunter [-h] --level LEVEL
```

10.1.1 Treasure Hunter game settings

--level The difficulty level. Must be between 1 and 30 (included).

CHAPTER

ELEVEN

TEXTWORLD

11.1 Core

exception textworld.core.**EnvInfoMissingError**(*requester*, *info*)

Bases: NameError

Thrown whenever some environment information EnvInfos.

exception textworld.core.**GameNotRunningError**(*msg=''*)

Bases: RuntimeError

Error when game is not running (either has terminiated or crashed).

class textworld.core.**Agent**

Bases: object

Interface for any agent that want to play a text-based game.

act(*game_state*, *reward*, *done*)

Acts upon the current game state.

Parameters

- **game_state** (*GameState*) – Current game state.
- **reward** (float) – Accumulated reward up until now.
- **done** (bool) – Whether the game is finished.

Return type
str

Returns
Text command to be performed in this current state.

finish(*game_state*, *reward*, *done*)

Let the agent know the game has finished.

Parameters

- **game_state** (*GameState*) – Game state at the moment the game finished.
- **reward** (float) – Accumulated reward up until now.
- **done** (bool) – Whether the game has finished normally or not. If False, it means the agent's used up all of its actions.

Return type
None

reset(*env*)

Let the agent set some environment's flags.

Parameters
env (*Environment*) – TextWorld environment.

Return type
None

property wrappers

class textworld.core.**EnvInfos**(***kwargs*)

Bases: object

Customizing what information will be returned by an environment.

Information can be requested by setting one or more attributes to True. The attribute *extras* should be a list of strings corresponding to keys in the metadata dictionary of TextWorld generated games.

copy()

admissible_commands

All commands relevant to the current state. This information changes from one step to another.

Type
bool

property basics: Iterable[str]

Information requested excluding the extras.

Return type
Iterable[str]

command_templates

Templates for commands understood by the the game. This information *doesn't* change from one step to another.

Type
bool

description

Text description of the current room, i.e. output of the `look` command. This information changes from one step to another.

Type
bool

entities

Names of all entities in the game. This information *doesn't* change from one step to another.

Type
bool

extras

Names of extra information which are game specific.

Type
List[str]

`facts`

All the facts that are currently true about the world. This information changes from one step to another.

Type
bool

`fail_facts`

Mutually exclusive sets of failing facts for each quest. This information *doesn't* change from one step to another.

Type
bool

`feedback`

Text observation produced by the game in response to previous command. This information changes from one step to another.

Type
bool

`game`

Current game in its serialized form. Use with `textworld.Game.deserialize`.

Type
bool

`intermediate_reward`

Reward (proxy) indicating if the player is making progress. This information changes from one step to another.

Type
bool

`inventory`

Text listing of the player's inventory, i.e. output of the *`inventory`* command. This information changes from one step to another.

Type
bool

`last_action`

The last action performed where `None` means it was not a valid action. This information changes from one step to another.

Type
bool

`last_command`

The last command performed where `None` means it was not a valid command. This information changes from one step to another.

Type
bool

`location`

Name of the player's current location. This information changes from one step to another.

Type
bool

`lost`

Whether the player lost the game. This information changes from one step to another.

Type
bool

`max_score`

Maximum reachable score of the game. This information *doesn't* change from one step to another.

Type
bool

`moves`

Number of moves done so far in the game. This information changes from one step to another.

Type
bool

`objective`

Objective of the game described in text. This information *doesn't* change from one step to another.

Type
bool

`policy_commands`

Sequence of commands leading to a winning state. This information changes from one step to another.

Type
bool

`score`

Current score of the game. This information changes from one step to another.

Type
bool

`verbs`

Verbs understood by the the game. This information *doesn't* change from one step to another.

Type
bool

`win_facts`

Mutually exclusive sets of winning facts for each quest. This information *doesn't* change from one step to another.

Type
bool

`won`

Whether the player won the game. This information changes from one step to another.

Type
bool

`class textworld.core.Environment(infos=None)`

Bases: `object`

Class allowing to interact with the game's interpreter.

The role of an *Environment* is to handle the communication between user code and the backend interpreter that manages the text-based game. The overall *Environment* structure is highly inspired by OpenAI's gym.

Example

Here's a minimal example of how to interact with an *Environment*

```
>>> import textworld
>>> options = textworld.GameOptions()
>>> options.seeds = 1234
>>> options.nb_objects = 5
>>> options.quest_length = 2
>>> game_file, _ = textworld.make(options, path='./')  # Generate a random game.
>>> env = textworld.start(game_file)  # Load the game.
>>> game_state = env.reset()  # Start a new game.
>>> env.render()
I hope you're ready to go into rooms and interact with objects, because you've
just entered TextWorld! Here is how to play! First thing I need you to do is to
ensure that the type G chest is open. And then, pick up the keycard from the
type G chest inside the attic. Got that? Good!

-= Attic =-
You arrive in an attic. A normal kind of place. You begin to take stock of
what's in the room.

You make out a type G chest. You can see a TextWorld style locker. The TextWorld
style locker contains a frisbee and a sock.



There is a TextWorld style key on the floor.
>>> command = "take key"  # Command to send to the game.
>>> game_state, reward, done = env.step(command)
>>> env.render()
(the TextWorld style key)
You pick up the TextWorld style key from the ground.
```

Parameters
infos (Optional[*EnvInfos*]) – Information to be included in the game state. By default, only the game's narrative is included.

close()

Ends the game.

Return type
None

copy()

Return a copy of this environment at the same state.

Return type
Environment

Returns
A copy of this environment at the same state.

load(*path*)

Loads a new text-based game.

Parameters
path (str) – Path to the game file to load.

Return type
None

render(*mode='human'*)

Renders the current state of the game.

Parameters
mode (str) – The mode to use for rendering.

Return type
Optional[str]

reset()

Starts game from the beginning.

Return type
GameState

Returns
Initial state of the game.

seed(*seed=None*)

Sets the seed for the random number generator.

Return type
None

step(*command*)

Performs a given command.

Parameters
command (str) – Text command to send to the interpreter.

Return type
Tuple[*GameState*, float, bool]

Returns
A tuple containing the new game state, a reward for performing that command and reaching this new state, and whether the game is finished or not.

property display_command_during_render: bool

Enables/disables displaying the command when rendering.

Return type
bool

class textworld.core.**GameState**

Bases: dict

copy()

Returns a deepcopy of this game state.

Return type
GameState

class textworld.core.**Wrapper**(*env=None*)

Bases: object

Special environment that wraps others to provide new functionalities.

Special environment that wraps other *Environment* objects to provide new functionalities (e.g. transcript recording, viewer, etc).

Parameters
env (Optional[*Environment*]) – environment to wrap.

close()

Return type
None

copy()

Return type
Wrapper

load(*path*)

Return type
None

render(*mode='human'*)

Return type
Optional[Any]

reset()

Return type
GameState

seed(*seed=None*)

Return type
List[int]

step(*command*)

Return type
Tuple[*GameState*, float, bool]

property display_command_during_render: bool

Return type
bool

property unwrapped

CHAPTER

TWELVE

TEXTWORLD.GYM

textworld.gym.utils.**register_game**(*gamefile*, *request_infos=None*, *batch_size=None*, *auto_reset=False*, *max_episode_steps=50*, *asynchronous=True*, *action_space=None*, *observation_space=None*, *name=''*, ***kwargs*)

Make an environment for a particular game.

Parameters

- **gamefile** (str) – Path for the TextWorld game (*.ulx|*.z[1-8]).
- **request_infos** (Optional[*EnvInfos*]) – For customizing the information returned by this environment (see textworld.EnvInfos for the list of available information).

  Warning: Only supported for TextWorld games (i.e., with a corresponding *.json file).

- **batch_size** (Optional[int]) – If provided, it indicates the number of games to play at the same time. By default, a single game is played at once.

  Warning: When batch_size is provided (even for batch_size=1), env.step expects a list of commands as input and outputs a list of states. env.reset also outputs a list of states.

- **auto_reset** (bool) – If True, each game *independently* resets once it is done (i.e., reset happens on the next env.step call). Otherwise, once a game is done, subsequent calls to env.step won't have any effects.
- **max_episode_steps** (int) – Number of steps allocated to play each game. Once exhausted, the game is done.
- **asynchronous** (bool) – If True, games in the batch are played in parallel. Only when batch size is greater than one.
- **action_space** (Optional[Space]) – The action space be used with OpenAI baselines. (see *textworld.gym.spaces.Word*).
- **observation_space** (Optional[Space]) – The observation space be used with OpenAI baselines (see *textworld.gym.spaces.Word*).
- **name** (str) – Name for the new environment, i.e. "tw-{name}-v0". By default, the returned env_id is "tw-v0".

Return type

str

Returns
The corresponding gym-compatible env_id to use.

Example

```
>>> from textworld.generator import make_game, compile_game
>>> options = textworld.GameOptions()
>>> options.seeds = 1234
>>> game = make_game(options)
>>> game.extras["more"] = "This is extra information."
>>> gamefile = compile_game(game)

>>> import gym
>>> import textworld.gym
>>> from textworld import EnvInfos
>>> request_infos = EnvInfos(description=True, inventory=True, extras=["more"])
>>> env_id = textworld.gym.register_game(gamefile, request_infos)
>>> env = gym.make(env_id)
>>> ob, infos = env.reset()
>>> print(infos["extra.more"])
This is extra information.
```

textworld.gym.utils.**register_games**(*gamefiles*, *request_infos=None*, *batch_size=None*, *auto_reset=False*, *max_episode_steps=50*, *asynchronous=True*, *action_space=None*, *observation_space=None*, *name=''*, ***kwargs*)

Make an environment that will cycle through a list of games.

Parameters

- **gamefiles** (List[str]) – Paths for the TextWorld games (*.ulx|*.z[1-8]).
- **request_infos** (Optional[*EnvInfos*]) – For customizing the information returned by this environment (see textworld.EnvInfos for the list of available information).

  Warning: Only supported for TextWorld games (i.e., with a corresponding *.json file).

- **batch_size** (Optional[int]) – If provided, it indicates the number of games to play at the same time. By default, a single game is played at once.

  Warning: When batch_size is provided (even for batch_size=1), env.step expects a list of commands as input and outputs a list of states. env.reset also outputs a list of states.

- **auto_reset** (bool) – If True, each game *independently* resets once it is done (i.e., reset happens on the next env.step call). Otherwise, once a game is done, subsequent calls to env.step won't have any effects.
- **max_episode_steps** (int) – Number of steps allocated to play each game. Once exhausted, the game is done.
- **asynchronous** (bool) – If True, games in the batch are played in parallel. Only when batch size is greater than one.

- **action_space** (Optional[Space]) – The action space be used with OpenAI baselines. (see *textworld.gym.spaces.Word*).
- **observation_space** (Optional[Space]) – The observation space be used with OpenAI baselines (see *textworld.gym.spaces.Word*).
- **name** (str) – Name for the new environment, i.e. "tw-{name}-v0". By default, the returned env_id is "tw-v0".

Return type
str

Returns
The corresponding gym-compatible env_id to use.

Example

```
>>> from textworld.generator import make_game, compile_game
>>> options = textworld.GameOptions()
>>> options.seeds = 1234
>>> game = make_game(options)
>>> game.extras["more"] = "This is extra information."
>>> gamefile = compile_game(game)

>>> import gym
>>> import textworld.gym
>>> from textworld import EnvInfos
>>> request_infos = EnvInfos(description=True, inventory=True, extras=["more"])
>>> env_id = textworld.gym.register_games([gamefile], request_infos)
>>> env = gym.make(env_id)
>>> ob, infos = env.reset()
>>> print(infos["extra.more"])
This is extra information.
```

12.1 Agent

class textworld.gym.core.**Agent**

Bases: object

Interface for any agent playing TextWorld games.

act(*obs*, *score*, *done*, *infos*)

Acts upon the current list of observations.

One text command must be returned for each observation.

Parameters

- **obs** (str) – Previous command's feedback (game's narrative).
- **score** (int) – The score obtained so far.
- **done** (bool) – Whether the game is finished.
- **infos** (Mapping[str, Any]) – Additional information requested.

Return type
str

Returns
Text command to be performed. If episode has ended (i.e. `done` is `True`), the returned value is expected to be ignored.

property infos_to_request: *EnvInfos*

Returns what additional information should be made available at each game step.

Requested information will be included within the `infos` dictionary passed to *Agent.act()*. To request specific information, create a `textworld.EnvInfos` and set its attributes to `True` accordingly.

In addition to the standard information, certain games may have specific information that can be requested via the `extras` attribute. Refer to the documentation specific to the game to know more (see `textworld.challenges`).

Example

Here is an example of how to request information and retrieve it.

```
>>> from textworld import EnvInfos
>>> request_infos = EnvInfos(description=True, inventory=True)
...
>>> env = gym.make(env_id)
>>> ob, infos = env.reset()
>>> print(infos["description"])
>>> print(infos["inventory"])
```

Return type
EnvInfos

12.2 Envs

class textworld.gym.envs.textworld.**TextworldGymEnv**(*gamefiles*, *request_infos=None*, *max_episode_steps=None*, *action_space=None*, *observation_space=None*, ***kwargs*)

Bases: *TextworldBatchGymEnv*

Environment for playing text-based games.

Parameters

- **gamefiles** (`List[str]`) – Paths of every game composing the pool (`*.ulx|*.z[1-8]`).
- **request_infos** (`Optional[EnvInfos]`) – For customizing the information returned by this environment (see `textworld.EnvInfos` for the list of available information).

  Warning: Only supported for TextWorld games (i.e., that have a corresponding `*.json` file).

- **max_episode_steps** (`Optional[int]`) – Number of steps allocated to play each game. Once exhausted, the game is done.

- **action_space** (Optional[Space]) – The action space be used with OpenAI baselines. (see *textworld.gym.spaces.Word*).
- **observation_space** (Optional[Space]) – The observation space be used with OpenAI baselines (see *textworld.gym.spaces.Word*).

reset()

Resets the text-based environment.

Resetting this environment means starting the next game in the pool.

Return type
Tuple[str, Dict[str, Any]]

Returns

A tuple (observation, info) where

- observation: text observed in the initial state;
- infos: additional information as requested.

step(*command*)

Runs a command in the text-based environment.

Parameters
command – Text command to send to the game interpreter.

Return type
Tuple[str, Dict[str, Any]]

Returns

A tuple (observation, score, done, info) where

- observation: text observed in the new state;
- score: total number of points accumulated so far;
- done: whether the game is finished or not;
- infos: additional information as requested.

action_space: Space[ActType]

metadata: Dict[str, Any] = {'render.modes': ['human', 'ansi', 'text']}

observation_space: Space[ObsType]

class textworld.gym.envs.textworld_batch.**TextworldBatchGymEnv**(*gamefiles*, *request_infos=None*, *batch_size=1*, *asynchronous=True*, *auto_reset=False*, *max_episode_steps=None*, *action_space=None*, *observation_space=None*)

Bases: Env

Environment for playing text-based games in batch.

Parameters

- **gamefiles** (List[str]) – Paths of every game composing the pool (*.ulx|*.z[1-8]|*.json).

- **request_infos** (Optional[*EnvInfos*]) – For customizing the information returned by this environment (see `textworld.EnvInfos` for the list of available information).

  Warning: Only supported for TextWorld games (i.e., that have a corresponding `*.json` file).

- **batch_size** (int) – If provided, it indicates the number of games to play at the same time. By default, a single game is played at once.

  Warning: When `batch_size` is provided (even for batch_size=1), `env.step` expects a list of commands as input and outputs a list of states. `env.reset` also outputs a list of states.

- **asynchronous** (bool) – If `True`, wraps the environments in an `AsyncBatchEnv` (which uses `multiprocessing` to run the environments in parallel). If `False`, wraps the environments in a `SyncBatchEnv`. Default: `True`.
- **auto_reset** (bool) – If `True`, each game *independently* resets once it is done (i.e., reset happens on the next `env.step` call). Otherwise, once a game is done, subsequent calls to `env.step` won't have any effects.
- **max_episode_steps** (Optional[int]) – Number of steps allocated to play each game. Once exhausted, the game is done.
- **action_space** (Optional[Space]) – The action space be used with OpenAI baselines. (see *textworld.gym.spaces.Word*).
- **observation_space** (Optional[Space]) – The observation space be used with OpenAI baselines (see *textworld.gym.spaces.Word*).

close()

Close this environment.

Return type
None

render(**args: Tuple[Any]*, ***kwargs: Dict[str, Any]*) → Optional[Union[RenderFrame, List[RenderFrame]]]

Compute the render frames as specified by render_mode attribute during initialization of the environment.

The set of supported modes varies per environment. (And some third-party environments may not support rendering at all.) By convention, if render_mode is:

- None (default): no render is computed.
- human: render return None. The environment is continuously rendered in the current display or terminal. Usually for human consumption.
- single_rgb_array: return a single frame representing the current state of the environment. A frame is a numpy.ndarray with shape (x, y, 3) representing RGB values for an x-by-y pixel image.
- rgb_array: return a list of frames representing the states of the environment since the last reset. Each frame is a numpy.ndarray with shape (x, y, 3), as with single_rgb_array.
- ansi: Return a list of strings (str) or StringIO.StringIO containing a terminal-style text representation for each time step. The text can include newlines and ANSI escape sequences (e.g. for colors).

Note: Rendering computations is performed internally even if you don't call render(). To avoid this, you can set render_mode = None and, if the environment supports it, call render() specifying the argument 'mode'.

Note: Make sure that your class's metadata 'render_modes' key includes the list of supported modes. It's recommended to call super() in implementations to use the functionality of this method.

Return type
Union[TypeVar(RenderFrame), List[TypeVar(RenderFrame)], None]

reset()

Resets the text-based environment.

Resetting this environment means starting the next game in the pool.

Return type
Tuple[List[str], Dict[str, List[Any]]]

Returns

A tuple (observations, infos) where

- observation: text observed in the initial state for each game in the batch;
- infos: additional information as requested for each game in the batch.

seed(*seed=None*)

Set the seed for this environment's random generator(s).

This environment use a random generator to shuffle the order in which the games are played.

Parameters
seed (Optional[int]) – Number that will be used to seed the random generators.

Return type
List[int]

Returns
All the seeds used to set this environment's random generator(s).

skip(*nb_games=1*)

Skip games.

Parameters
nb_games (int) – Number of games to skip.

Return type
None

step(*commands*)

Runs a command in each text-based environment of the batch.

Parameters
commands – Text command to send to the game interpreter.

Return type
Tuple[List[str], List[float], List[bool], Dict[str, List[Any]]]

Returns

A tuple (observations, scores, dones, infos) where

- observations: text observed in the new state for each game in the batch;
- scores: total number of points accumulated so far for each game in the batch;
- dones: whether each game in the batch is finished or not;
- infos: additional information as requested for each game in the batch.

action_space: Space[ActType]

metadata: Dict[str, Any] = {'render.modes': ['human', 'ansi', 'text']}

observation_space: Space[ObsType]

textworld.gym.envs.utils.**shuffled_cycle**(*iterable*, *rng*, *nb_loops=-1*)

Yield each element of `iterable` one by one, then shuffle the elements and start yielding from the start. Stop after `nb_loops` loops.

Parameters

- **iterable** (`Iterable[Any]`) – Iterable containing the elements to yield.
- **rng** (`RandomState`) – Random generator used to shuffle the elements after each loop.
- **nb_loops** (`int`) – Number of times to go through all the elements. If set to -1, loop an infinite number of times.

Return type
`Iterable[Any]`

12.3 Spaces

exception textworld.gym.spaces.text_spaces.**VocabularyHasDuplicateTokens**

Bases: `ValueError`

class textworld.gym.spaces.text_spaces.**Char**(*max_length*, *vocab=None*, *extra_vocab=[]*)

Bases: `MultiDiscrete`

Character observation/action space

This space consists of a series of `gym.spaces.Discrete` objects all with the same parameters. Each `gym.spaces.Discrete` can take integer values between 0 and len(self.vocab).

Notes

The following special token will be prepended (if needed) to the vocabulary:

- '#' : Padding token

Parameters

- **max_length** (*int*) – Maximum number of characters in a text.
- **vocab** (*list of char, optional*) – Vocabulary defining this space. It shouldn't contain any duplicate characters. If not provided, the vocabulary will consists in characters [a-z0-9], punctuations [" ", "-", "'"] and padding '#'.

- **extra_vocab** (*list of char, optional*) – Additional tokens to add to the vocabulary.

filter_unknown(*text*)

Strip out all characters not in the vocabulary.

tokenize(*text*, *padding=False*)

Tokenize characters found in the vocabulary.

Note: text will be padded up to `self.max_length`.

class textworld.gym.spaces.text_spaces.**Word**(*max_length*, *vocab*)

Bases: `MultiDiscrete`

Word observation/action space

This space consists of a series of `gym.spaces.Discrete` objects all with the same parameters. Each `gym.spaces.Discrete` can take integer values between 0 and `len(self.vocab)`.

Notes

The following special tokens will be prepended (if needed) to the vocabulary:

- '<PAD>' : Padding
- '<UNK>' : Unknown word
- '<S>' : Beginning of sentence
- '</S>' : End of sentence

Example

Let's create an action space that can be used with *textworld.gym.register_game*. We are going to assume actions are short phrases up to 8 words long.

```
>>> import textworld
>>> gamefiles = ["/path/to/game.ulx", "/path/to/another/game.z8"]
>>> vocab = textworld.vocab.extract_from(gamefiles)
>>> vocab = sorted(vocab)  # Sorting the vocabulary, optional.
>>> action_space = textworld.gym.text_spaces.Word(max_length=8, vocab=vocab)
```

Parameters

- **max_length** (*int*) – Maximum number of words in a text.
- **vocab** (*list of strings*) – Vocabulary defining this space. It shouldn't contain any duplicate words.

tokenize(*text*, *padding=False*)

Tokenize words found in the vocabulary.

Note: text will be padded up to `self.max_length`.

CHAPTER

THIRTEEN

TEXTWORLD.ENVS

13.1 TextWorld

class textworld.envs.tw.**TextWorldEnv**(*infos=None*)

Bases: *Environment*

Environment for playing games by TextWorld.

Parameters
infos (Optional[*EnvInfos*]) – Information to be included in the game state. By default, only the game's narrative is included.

copy()

Return a copy of this environment.

It is safe to call *step* and *reset* on the copied environment.

> **Warning:** The Game and Inform7Game private objects are *soft* copies.

Return type
TextWorldEnv

load(*path*)

Loads a new text-based game.

Parameters
path (str) – Path to the game file to load.

Return type
None

reset()

Starts game from the beginning.

Returns
Initial state of the game.

step(*command*)

Performs a given command.

Parameters
command (str) – Text command to send to the interpreter.

Returns
A tuple containing the new game state, a reward for performing that command and reaching this new state, and whether the game is finished or not.

13.2 Glulx

class textworld.envs.glulx.git_glulx.**GitGlulxEnv**(**args*, ***kwargs*)

Bases: *Environment*

Environment to support playing Glulx games.

This environment supports playing text-based games that were compiled for the Glulx virtual machine. The main advantage of using Glulx over Z-Machine is it uses 32-bit data and addresses, so it can handle game files up to four gigabytes long. This comes handy when we want to generate large world with a lot of objects in it.

We use a customized version of git-glulx as the glulx interpreter. That way we don't rely on stdin/stdout to communicate with the interpreter but instead use UNIX sockets.

Parameters
infos – Information to be included in the game state. By default, only the game's narrative is included.

close()

Ends the game.

Return type
None

load(*ulx_file*)

Loads a new text-based game.

Parameters
path – Path to the game file to load.

Return type
None

render(*mode='human'*)

Renders the current state of the game.

Parameters
mode (str) – The mode to use for rendering.

Return type
None

reset()

Starts game from the beginning.

Return type
str

Returns
Initial state of the game.

step(*command*)

Performs a given command.

Parameters
command (str) – Text command to send to the interpreter.

Return type
str

Returns
A tuple containing the new game state, a reward for performing that command and reaching this new state, and whether the game is finished or not.

property game_running: bool

Determines if the game is still running.

Return type
bool

13.3 Wrappers

class textworld.envs.wrappers.recorder.**Recorder**

Bases: *Wrapper*

Parameters
env – environment to wrap.

reset()

Return type
GameState

step(*command*)

Return type
Tuple[*GameState*, float, bool]

class textworld.envs.wrappers.viewer.**HtmlViewer**(*env*, *open_automatically=True*, *port=8080*)

Bases: *Wrapper*

Wrap a TextWorld environment to provide visualization.

During a playthrough, the game can be visualized via local webserver http://localhost:<port>.

:param : The TextWorld environment to wrap. :type : param env: :param : Port to use for the web viewer. :type : param port:

close()

Close the game.

In addition to shutting down the game, this closes the local webserver.

reset()

Reset the game.

Return type
Initial game state.

step(*command*)

Perform a game step.

Parameters
command (str) – Text command to send to the game engine.

Return type
Tuple[*GameState*, float, bool]

Returns

- *game_state* – Updated game state.
- *score* – Score for reaching this state.
- *done* – Whether the same is done or not.

property port

class textworld.envs.wrappers.filter.**Filter**(*env=None*)

Bases: *Wrapper*

Environment wrapper to filter what information is made available.

Requested information will be included within the `infos` dictionary returned by *Filter.reset()* and `Filter.step(...)`. To request specific information, create a `textworld.EnvInfos` and set the appropriate attributes to `True`. Then, instantiate a *Filter* wrapper with the `EnvInfos` object.

Example

Here is an example of how to request information and retrieve it.

```
>>> from textworld import EnvInfos
>>> from textworld.envs.wrappers import Filter
>>> request_infos = EnvInfos(description=True, inventory=True, extras=["more"])
>>> env = textworld.start(gamefile, request_infos)
>>> env = Filter(env)
>>> ob, infos = env.reset()
>>> print(infos["description"])
>>> print(infos["inventory"])
>>> print(infos["extra.more"])
```

Parameters
env (Optional[*Environment*]) – environment to wrap.

copy()

Return type
Filter

reset()

Return type
Tuple[str, Mapping[str, Any]]

step(*command*)

Return type
Tuple[str, Mapping[str, Any]]

13.4 Z-Machine

class textworld.envs.zmachine.jericho.**JerichoEnv**(**args*, ***kwargs*)

Bases: *Environment*

Parameters
infos – Information to be included in the game state. By default, only the game's narrative is included.

close()

Ends the game.

copy()

Return a copy of this environment at the same state.

Return type
JerichoEnv

load(*z_file*)

Loads a new text-based game.

Parameters
path – Path to the game file to load.

Return type
None

reset()

Starts game from the beginning.

Returns
Initial state of the game.

seed(*seed=None*)

Sets the seed for the random number generator.

step(*command*)

Performs a given command.

Parameters
command – Text command to send to the interpreter.

Returns
A tuple containing the new game state, a reward for performing that command and reaching this new state, and whether the game is finished or not.

property game_running: bool

Determines if the game is still running.

Return type
bool

CHAPTER

FOURTEEN

TEXTWORLD.AGENTS

class textworld.agents.human.**HumanAgent**(*autocompletion=True*, *oracle=False*)

Bases: *Agent*

act(*game_state*, *reward*, *done*)

Acts upon the current game state.

Parameters

- **game_state** – Current game state.
- **reward** – Accumulated reward up until now.
- **done** – Whether the game is finished.

Returns

Text command to be performed in this current state.

reset(*env*)

Let the agent set some environment's flags.

Parameters

env – TextWorld environment.

class textworld.agents.random.**NaiveAgent**(*seed=1234*)

Bases: *Agent*

act(*game_state*, *reward*, *done*)

Acts upon the current game state.

Parameters

- **game_state** – Current game state.
- **reward** – Accumulated reward up until now.
- **done** – Whether the game is finished.

Returns

Text command to be performed in this current state.

reset(*env*)

Let the agent set some environment's flags.

Parameters

env – TextWorld environment.

class textworld.agents.random.**RandomCommandAgent**(*seed=1234*)

Bases: *Agent*

act(*game_state*, *reward*, *done*)

Acts upon the current game state.

Parameters

- **game_state** – Current game state.
- **reward** – Accumulated reward up until now.
- **done** – Whether the game is finished.

Returns

Text command to be performed in this current state.

reset(*env*)

Let the agent set some environment's flags.

Parameters

env – TextWorld environment.

class textworld.agents.simple.**NaiveAgent**(*seed=1234*)

Bases: *Agent*

act(*game_state*, *reward*, *done*)

Acts upon the current game state.

Parameters

- **game_state** – Current game state.
- **reward** – Accumulated reward up until now.
- **done** – Whether the game is finished.

Returns

Text command to be performed in this current state.

reset(*env*)

Let the agent set some environment's flags.

Parameters

env – TextWorld environment.

exception textworld.agents.walkthrough.**WalkthroughDone**

Bases: NameError

class textworld.agents.walkthrough.**WalkthroughAgent**(*commands=None*)

Bases: *Agent*

Agent that simply follows a list of commands.

act(*game_state*, *reward*, *done*)

Acts upon the current game state.

Parameters

- **game_state** – Current game state.
- **reward** – Accumulated reward up until now.
- **done** – Whether the game is finished.

Returns

Text command to be performed in this current state.

reset(*env*)

Let the agent set some environment's flags.

Parameters

env – TextWorld environment.

CHAPTER

FIFTEEN

TEXTWORLD.GENERATOR

exception textworld.generator.**GenerationWarning**

Bases: UserWarning

textworld.generator.**compile_game**(*game*, *options=None*)

Compile a game.

Parameters

- **game** (*Game*) – Game object to compile.
- **options** (Optional[*GameOptions*]) – For customizing the game generation (see *textworld.GameOptions* for the list of available options).

Returns

The path to compiled game.

textworld.generator.**make_game**(*options*)

Make a game (map + objects + quest).

Parameters

options (*GameOptions*) – For customizing the game generation (see *textworld.GameOptions* for the list of available options).

Return type

Game

Returns

Generated game.

textworld.generator.**make_game_with**(*world*, *quests=None*, *grammar=None*)

textworld.generator.**make_grammar**(*options={}*, *rng=None*)

Return type

Grammar

textworld.generator.**make_map**(*n_rooms*, *size=None*, *rng=None*, *possible_door_states=['open', 'closed', 'locked']*)

Make a map.

Parameters

- **n_rooms** (*int*) – Number of rooms in the map.
- **size** (*tuple of int*) – Size (height, width) of the grid delimiting the map.

textworld.generator.**make_quest**(*world*, *options=None*)

textworld.generator.**make_small_map**(*n_rooms*, *rng=None*, *possible_door_states=['open', 'closed', 'locked']*)

Make a small map.

The map will contains one room that connects to all others.

Parameters

- **n_rooms** (*int*) – Number of rooms in the map (maximum of 5 rooms).
- **possible_door_states** (*list of str, optional*) – Possible states doors can have.

textworld.generator.**make_world**(*world_size*, *nb_objects=0*, *rngs=None*)

Make a world (map + objects).

Parameters

- **world_size** (*int*) – Number of rooms in the world.
- **nb_objects** (*int*) – Number of objects in the world.

textworld.generator.**make_world_with**(*rooms*, *rng=None*)

Make a world that contains the given rooms.

Parameters

rooms (*list of textworld.logic.Variable*) – Rooms in the map. Variables must have type 'r'.

exception textworld.generator.chaining.**QuestGenerationError**

Bases: Exception

class textworld.generator.chaining.**Chain**(*initial_state*, *nodes*)

Bases: object

An initial state and a chain of actions forming a quest.

nodes

The dependency tree of this quest.

initial_state

The initial state from which the actions start.

actions

The sequence of actions forming this quest.

class textworld.generator.chaining.**ChainNode**(*action*, *depth*, *breadth*, *parent*)

Bases: object

A node in a chain of actions.

action

The action to perform at this step.

depth

This node's depth in the dependency tree.

breadth

This node's breadth in the dependency tree.

parent

This node's parent in the dependency tree.

class textworld.generator.chaining.**ChainingOptions**

Bases: object

Options for customizing the behaviour of chaining.

backward

Whether to run chaining forwards or backwards. Forward chaining produces a sequence of actions that start at the provided state, while backward chaining produces a sequence of actions that end up at the provided state.

min_length

The minimum length of the generated quests.

max_length

The maximum length of the generated quests.

min_depth

The minimum depth (length) of the generated independent subquests.

max_depth

The maximum depth (length) of the generated independent subquests.

min_breadth

The minimum breadth of the generated quests. When this is higher than 1, the generated quests will have multiple parallel subquests. In this case, min_depth and max_depth limit the length of these independent subquests, not the total size of the quest.

max_breadth

The maximum breadth of the generated quests.

subquests

Whether to also return incomplete quests, which could be extended without reaching the depth or breadth limits.

independent_chains

Whether to allow totally independent parallel chains.

create_variables

Whether new variables may be created during chaining.

rng

If provided, randomize the order of the quests using this random number generator.

rules_per_depth

A list of lists of rules for restricting the allowed actions at certain depths.

restricted_types

A set of types that may not have new variables created.

allowed_types

A set of types that are allowed to have new variables created.

check_action(*state*, *action*)

Check if an action should be allowed in this state.

The default implementation disallows actions that would create new facts that don't mention any new variables.

Parameters

- **state** (*State*) – The current state.
- **action** (*Action*) – The action being applied.

Return type
bool

Returns
Whether that action should be allowed.

check_new_variable(*state*, *type*, *count*)

Check if a new variable should be allowed to be created in this state.

Parameters

- **state** (*State*) – The current state.
- **type** (str) – The type of variable being created.
- **count** (int) – The total number of variables of that type.

Return type
bool

Returns
Whether that variable should be allowed to be created.

copy()

Return type
ChainingOptions

get_rules(*depth*)

Get the relevant rules for this depth.

Parameters
depth (int) – The current depth in the chain.

Return type
Iterable[*Rule*]

Returns
The rules that may be applied at this depth in the chain.

property fixed_mapping: *GameLogic*

A fixed mapping from placeholders to variables, for singletons.

Return type
GameLogic

property logic: *GameLogic*

The rules of the game.

Return type
GameLogic

textworld.generator.chaining.**get_chains**(*state*, *options*)

Generates chains of actions (quests) starting from or ending at the given state.

Parameters

- **state** (*State*) – The initial state for chaining.
- **options** (*ChainingOptions*) – Options to configure chaining behaviour.

Return type
Iterable[*Chain*]

Returns
All possible quests according to the constraints.

textworld.generator.chaining.**sample_quest**(*state*, *options*)

Samples a single chain of actions (a quest) starting from or ending at the given state.

Parameters

- **state** (*State*) – The initial state for chaining.
- **options** (*ChainingOptions*) – Options to configure chaining behaviour. Set options.rng to sample a random quest.

Return type
Optional[*Chain*]

Returns
A single possible quest.

Raises
QuestGenerationError – No quest could be generated given the provided chaining options.

class textworld.generator.dependency_tree.**DependencyTree**(*element_type=<class 'textworld.generator.dependency_tree.DependencyTreeEleme*
trees=[])

Bases: object

copy()

Return type
DependencyTree

push(*value*, *allow_multi_root=False*)

Add a value to this dependency tree.

Adding a value already present in the tree does not modify the tree.

Parameters

- **value** (Any) – value to add.
- **allow_multi_root** (bool) – if True, allow the value to spawn an additional root if needed.

Return type
bool

remove(*value*)

Remove all leaves having the given value.

The value to remove needs to belong to at least one leaf in this tree. Otherwise, the tree remains unchanged.

Parameters
value (Any) – value to remove from the tree.

Return type
bool

Returns
Whether the tree has changed or not.

property empty: bool

Return type
bool

property leaves_elements: List[DependencyTreeElement]

Return type
List[DependencyTreeElement]

property leaves_values: List[Any]

Return type
List[Any]

property values: List[Any]

Return type
List[Any]

class textworld.generator.dependency_tree.DependencyTreeElement(*value*)

Bases: object

Representation of an element in the dependency tree.

The notion of dependency and ordering should be defined for these elements.

Subclasses should override *depends_on*, __lt__ and __str__ accordingly.

depends_on(*other*)

Check whether this element depends on the other.

Return type
bool

is_distinct_from(*others*)

Check whether this element is distinct from others.

Return type
bool

class textworld.generator.logger.GameLogger(*group_actions=True*)

Bases: object

aggregate(*other*)

collect(*game*)

display_stats()

static load(*filename*)

save(*filename*)

stats()

exception textworld.generator.vtypes.NotEnoughNounsError

Bases: NameError

class textworld.generator.vtypes.VariableType(*type*, *name*, *parent=None*)

Bases: object

classmethod deserialize(*data*)

Return type
VariableType

classmethod parse(*expr*)

Parse a variable type expression.

Parameters
expr (str) – The string to parse, in the form `name: type -> parent1 & parent2` or `name: type` for root node.

Return type
VariableType

serialize()

Return type
str

class textworld.generator.vtypes.**VariableTypeTree**(*vtypes*)

Bases: object

Manages hierarchy of types defined in ./grammars/variables.txt. Used for extending the rules.

count(*state*)

Counts how many objects there are of each type.

descendants(*vtype*)

Given a variable type, return all possible descendants.

classmethod deserialize(*data*)

Return type
VariableTypeTree

get_ancestors(*vtype*)

List all ancestors of a type where the closest ancetors are first.

get_description(*vtype*)

is_constant(*vtype*)

is_descendant_of(*child*, *parents*)

Return if child is a descendant of parent

classmethod load(*path*)

Read variables from text file.

sample(*parent_type*, *rng*, *exceptions=[]*, *include_parent=True*, *probs=None*)

Sample an object type given the parent's type.

serialize()

Return type
List

CHEST = 'c'

CLASS_HOLDER = ['c', 's']

SUPPORTER = 's'

textworld.generator.vtypes.**get_new**(*type*, *types_counts*, *max_types_counts=None*)

Get the next available id for a given type.

textworld.generator.vtypes.**parse_variable_types**(*content*)

Parse a list VariableType expressions.

15.1 Game

exception textworld.generator.game.**UnderspecifiedEventError**

Bases: NameError

exception textworld.generator.game.**UnderspecifiedQuestError**

Bases: NameError

class textworld.generator.game.**ActionDependencyTree**(**args*, *kb=None*, ***kwargs*)

Bases: *DependencyTree*

copy()

Return type
ActionDependencyTree

flatten()

Generates a flatten representation of this dependency tree.

Actions are greedily yielded by iteratively popping leaves from the dependency tree.

Return type
Iterable[*Action*]

remove(*action*)

Remove all leaves having the given value.

The value to remove needs to belong to at least one leaf in this tree. Otherwise, the tree remains unchanged.

Parameters
value – value to remove from the tree.

Return type
Tuple[bool, Optional[*Action*]]

Returns
Whether the tree has changed or not.

class textworld.generator.game.**ActionDependencyTreeElement**(*value*)

Bases: *DependencyTreeElement*

Representation of an Action in the dependency tree.

The notion of dependency and ordering is defined as follows:

- action1 depends on action2 if action1 needs the propositions added by action2;
- action1 should be performed before action2 if action2 removes propositions needed by action1.

depends_on(*other*)

Check whether this action depends on the `other`.

Action1 depends on action2 when the intersection between the propositions added by action2 and the pre-conditions of the action1 is not empty, i.e. action1 needs the propositions added by action2.

Return type
`bool`

is_distinct_from(*others*)

Check whether this element is distinct from `others`.

We check if self.action has any additional information that `others` actions don't have. This helps us to identify whether a group of nodes in the dependency tree already contain all the needed information that self.action would bring.

Return type
`bool`

property action: *Action*

Return type
Action

class textworld.generator.game.**EntityInfo**(*id*, *type*)

Bases: `object`

Additional information about entities in the game.

classmethod deserialize(*data*)

Creates a *EntityInfo* from serialized data.

Parameters
data (`Mapping`) – Serialized data with the needed information to build a *EntityInfo* object.

Return type
EntityInfo

serialize()

Serialize this object.

Return type
`Mapping`

Returns
EntityInfo's data serialized to be JSON compatible.

adj

The adjective (i.e. descriptive) part of the name, if available.

Type
str

definite

The definite article to use for this entity.

Type
str

desc

Text description displayed when examining this entity in the game.

Type
str

`id`

Unique name for this entity. It is used when generating

Type
str

`indefinite`

The indefinite article to use for this entity.

Type
str

`name`

The name that will be displayed in-game to identify this entity.

Type
str

`noun`

The noun part of the name, if available.

Type
str

`room_type`

Type of the room this entity belongs to. It used to influence its *name* during text generation.

Type
str

`synonyms`

Alternative names that can be used to refer to this entity.

Type
List[str]

`type`

The type of this entity.

Type
str

`class textworld.generator.game.Event(actions=(), conditions=(), commands=())`

Bases: `object`

Event happening in TextWorld.

An event gets triggered when its set of conditions become all statisfied.

Parameters

- **actions** (`Iterable[Action]`) – The actions to be performed to trigger this event. If an empty list, then `conditions` must be provided.
- **conditions** (`Iterable[Proposition]`) – Set of propositions which need to be all true in order for this event to get triggered.
- **commands** (`Iterable[str]`) – Human readable version of the actions.

copy()

Copy this event.

Return type
Event

classmethod deserialize(*data*)

Creates an *Event* from serialized data.

Parameters
data (Mapping) – Serialized data with the needed information to build a *Event* object.

Return type
Event

is_triggering(*state*)

Check if this event would be triggered in a given state.

Return type
bool

serialize()

Serialize this event.

Results:
Event's data serialized to be JSON compatible.

Return type
Mapping

set_conditions(*conditions*)

Set the triggering conditions for this event.

Parameters
conditions (Iterable[*Proposition*]) – Set of propositions which need to be all true in order for this event to get triggered.

Return type
Action

Returns
Action that can only be applied when all conditions are statisfied.

property actions: Tuple[*Action*]

Actions to perform to trigger this event.

Return type
Tuple[*Action*]

property commands: Tuple[str]

Human readable version of the actions.

Return type
Tuple[str]

condition

Action that can only be applied when all conditions are statisfied.

Type
textworld.logic.Action

class textworld.generator.game.**EventProgression**(*event*, *kb*)

Bases: object

EventProgression monitors a particular event.

Internally, the event is represented as a dependency tree of relevant actions to be performed.

Parameters
quest – The quest to keep track of its completion.

compress_policy(*state*)

Compress the policy given a game state.

Parameters
state (*State*) – Current game state.

Return type
bool

Returns
Whether the policy was compressed or not.

copy()

Return a soft copy.

Return type
EventProgression

update(*action=None*, *state=None*)

Update event progression given available information.

Parameters

- **action** (Optional[*Action*]) – Action potentially affecting the event progression.
- **state** (Optional[*State*]) – Current game state.

Return type
None

property done: bool

Check if the quest is done (i.e. triggered or untriggerable).

Return type
bool

property triggered: bool

Check whether the event has been triggered.

Return type
bool

property triggering_policy: List[*Action*]

Actions to be performed in order to trigger the event.

Return type
List[*Action*]

property untriggerable: bool

Check whether the event is in an untriggerable state.

Return type
bool

class textworld.generator.game.**Game**(*world*, *grammar=None*, *quests=()*)

Bases: object

Game representation in TextWorld.

A *Game* is defined by a world and it can have quest(s) or not. Additionally, a grammar can be provided to control the text generation.

Parameters

- **world** (*World*) – The world to use for the game.
- **quests** (Iterable[*Quest*]) – The quests to be done in the game.
- **grammar** (Optional[*Grammar*]) – The grammar to control the text generation.

change_grammar(*grammar*)

Changes the grammar used and regenerate all text.

Return type
None

copy()

Make a shallow copy of this game.

Return type
Game

classmethod deserialize(*data*)

Creates a *Game* from serialized data.

Parameters
data (Mapping) – Serialized data with the needed information to build a *Game* object.

Return type
Game

classmethod load(*filename*)

Creates *Game* from serialized data saved in a file.

Return type
Game

save(*filename*)

Saves the serialized data of this game to a file.

Return type
None

serialize()

Serialize this object.

Results:
Game's data serialized to be JSON compatible

Return type
Mapping

property command_templates: List[str]

All command templates understood in this game.

Return type
List[str]

property directions_names: List[str]

Return type
List[str]

property entity_names: List[str]

Return type
List[str]

property infos: Dict[str, *EntityInfo*]

Information about the entities in the game.

Return type
Dict[str, *EntityInfo*]

property max_score: float

Sum of the reward of all quests.

Return type
float

property objective: str

Return type
str

property objects_names: List[str]

The names of all relevant objects in this game.

Return type
List[str]

property objects_names_and_types: List[str]

The names of all non-player objects along with their type in this game.

Return type
List[str]

property objects_types: List[str]

All types of objects in this game.

Return type
List[str]

property verbs: List[str]

Verbs that should be recognized in this game.

Return type
List[str]

property walkthrough: Optional[List[str]]

Return type
Optional[List[str]]

class textworld.generator.game.**GameOptions**

Bases: object

Options for customizing the game generation.

nb_rooms

Number of rooms in the game.

Type
int

nb_objects

Number of objects in the game.

Type
int

nb_parallel_quests

Number of parallel quests, i.e. not sharing a common goal.

Type
int

quest_depth

Number of actions that need to be performed to solve a subquest.

Type
int

path

Path of the compiled game (.ulx or .z8). Also, the source (.ni) and metadata (.json) files will be saved along with it.

Type
str

force_recompile

If True, recompile game even if it already exists.

Type
bool

file_ext

Type of the generated game file. Either .z8 (Z-Machine) or .ulx (Glulx). If *path* already has an extension, this is ignored.

Type
str

chaining

For customizing the quest generation (see *textworld.generator.ChainingOptions* for the list of available options).

Type
ChainingOptions

grammar

For customizing the text generation (see *textworld.generator.GrammarOptions* for the list of available options).

Type
GrammarOptions

`copy()`

Return type
GameOptions

`property kb: KnowledgeBase`

The knowledge base containing the logic and the text grammars (see *textworld.generator.KnowledgeBase* for more information).

Return type
KnowledgeBase

`property quest_breadth: int`

Number of subquests per independent quest. It controls how nonlinear a quest can be (1 means linear).

Return type
int

`property quest_length: int`

Number of actions that need to be performed to complete the game.

Return type
int

`property rngs: Dict[str, RandomState]`

Return type
Dict[str, RandomState]

`property seeds`

Seeds for the different generation processes.

- If None, seeds will be sampled from *textworld.g_rng*.
- If int, it acts as a seed for a random generator that will be used to sample the other seeds.
- If dict, the following keys can be set:
  - 'map': control the map generation;
  - 'objects': control the type of objects and their location;
  - 'quest': control the quest generation;
  - 'grammar': control the text generation.

  For any key missing, a random number gets assigned (sampled from *textworld.g_rng*).

`property uuid: str`

Return type
str

`class textworld.generator.game.GameProgression(game, track_quests=True)`

Bases: object

GameProgression keeps track of the progression of a game.

If *tracking_quests* is True, then *winning_policy* will be the list of Action that need to be applied in order to complete the game.

Parameters

- **game** (*Game*) – The game for which to track progression.
- **track_quests** (bool) – whether quest progressions are being tracked.

copy()

Return a soft copy.

Return type
GameProgression

update(*action*)

Update the state of the game given the provided action.

Parameters
action (*Action*) – Action affecting the state of the game.

Return type
None

property completed: bool

Whether all non-optional quests are completed.

Return type
bool

property done: bool

Whether all non-optional quests are completed or at least one has failed or is unfinishable.

Return type
bool

property failed: bool

Whether at least one non-optional quest has failed or is unfinishable.

Return type
bool

property score: int

Sum of the reward of all completed quests.

Return type
int

property tracking_quests: bool

Whether quests are being tracked or not.

Return type
bool

property valid_actions: List[*Action*]

Actions that are valid at the current state.

Return type
List[*Action*]

property winning_policy: Optional[List[*Action*]]

Actions to be performed in order to complete the game.

Return type
Optional[List[*Action*]]

Returns
A policy that leads to winning the game. It can be `None` if *tracking_quests* is `False` or the quest has failed.

class textworld.generator.game.**Quest**(*win_events=()*, *fail_events=()*, *reward=None*, *desc=None*, *commands=()*, *optional=False*, *repeatable=False*)

Bases: `object`

Quest representation in TextWorld.

A quest is defined by a mutually exclusive set of winning events and a mutually exclusive set of failing events.

Parameters

- **win_events** (`Iterable[Event]`) – Mutually exclusive set of winning events. That is, only one such event needs to be triggered in order to complete this quest.
- **fail_events** (`Iterable[Event]`) – Mutually exclusive set of failing events. That is, only one such event needs to be triggered in order to fail this quest.
- **reward** (`Optional[int]`) – Reward given for completing this quest. By default, reward is set to 1 if there is at least one winning events otherwise it is set to 0.
- **desc** (`Optional[str]`) – A text description of the quest.
- **commands** (`Iterable[str]`) – List of text commands leading to this quest completion.
- **optional** (`bool`) – If True, this quest is optional to finish the game.
- **repeatable** (`bool`) – If True, this quest can be completed more than once.

copy()

Copy this quest.

Return type
Quest

classmethod deserialize(*data*)

Creates a *Quest* from serialized data.

Parameters
data (`Mapping`) – Serialized data with the needed information to build a *Quest* object.

Return type
Quest

is_failing(*state*)

Check if this quest is failing in that particular state.

Return type
`bool`

is_winning(*state*)

Check if this quest is winning in that particular state.

Return type
`bool`

serialize()

Serialize this quest.

Return type
`Mapping`

Returns
Quest's data serialized to be JSON compatible.

property commands: Iterable[str]

List of text commands leading to this quest completion.

Return type
Iterable[str]

desc

A text description of the quest.

Type
str

property fail_events: Tuple[*Event*]

Mutually exclusive set of failing events. That is, only one such event needs to be triggered in order to fail this quest.

Return type
Tuple[*Event*]

optional

Whether this quest is optional or not to finish the game.

Type
bool

repeatable

Whether this quest can be completed more than once.

Type
bool

property win_events: Tuple[*Event*]

Mutually exclusive set of winning events. That is, only one such event needs to be triggered in order to complete this quest.

Return type
Tuple[*Event*]

class textworld.generator.game.**QuestProgression**(*quest*, *kb*)

Bases: object

QuestProgression keeps track of the completion of a quest.

Internally, the quest is represented as a dependency tree of relevant actions to be performed.

Parameters
quest (*Quest*) – The quest to keep track of its completion.

copy()

Return a soft copy.

Return type
QuestProgression

update(*action=None*, *state=None*)

Update quest progression given available information.

Parameters

- **action** (Optional[*Action*]) – Action potentially affecting the quest progression.
- **state** (Optional[*State*]) – Current game state.

Return type
None

property completable: bool

Check if the quest has winning events.

Return type
bool

property completed: bool

Check whether the quest is completed.

Return type
bool

property done: bool

Check if the quest is done (i.e. completed, failed or unfinishable).

Return type
bool

property failed: bool

Check whether the quest has failed.

Return type
bool

property unfinishable: bool

Check whether the quest is in an unfinishable state.

Return type
bool

property winning_policy: Optional[List[*Action*]]

Actions to be performed in order to complete the quest.

Return type
Optional[List[*Action*]]

textworld.generator.game.**gen_commands_from_actions**(*actions*, *kb=None*)

Return type
List[str]

15.2 World

exception textworld.generator.world.**NoFreeExitError**

Bases: Exception

class textworld.generator.world.**World**(*kb=None*)

Bases: object

add_fact(*fact*)

Return type
None

add_facts(*facts*)

Return type
None

classmethod deserialize(*serialized_facts*, *kb=None*)

Return type
World

find_object_by_id(*id*)

Return type
Optional[*WorldObject*]

find_room_by_id(*id*)

Return type
Optional[*WorldRoom*]

classmethod from_facts(*facts*, *kb=None*)

Return type
World

classmethod from_map(*map*, *kb=None*)

Parameters
map (Graph) – Graph defining the structure of the world.

Return type
World

get_all_objects_in(*obj*)

Return type
List[*WorldObject*]

get_entities_per_type(*type*)
Get all entities of a certain type.

Return type
List[*WorldEntity*]

get_facts_in_scope()

Return type
List[*Proposition*]

get_objects_in_inventory()

Return type
List[*WorldObject*]

get_visible_objects_in(*obj*)

Return type
List[*WorldObject*]

populate(*nb_objects*, *rng=None*, *object_types_probs=None*)

Return type
List[*Proposition*]

populate_room(*nb_objects*, *room*, *rng=None*, *object_types_probs=None*)

Return type
List[*Proposition*]

populate_room_with(*objects*, *room*, *rng=None*)

Return type
List[*Proposition*]

populate_with(*objects*, *rng=None*)

Return type
List[*Proposition*]

serialize()

Return type
List

set_player_room(*start_room=None*)

Return type
Proposition

property entities: ValuesView[*WorldEntity*]

Return type
ValuesView[*WorldEntity*]

property facts: List[*Proposition*]

Return type
List[*Proposition*]

property objects: List[*WorldObject*]

Return type
List[*WorldObject*]

property player_room: *WorldRoom*

Return type
WorldRoom

property rooms: List[*WorldRoom*]

Return type
List[*WorldRoom*]

property state: *State*

Return type
State

class textworld.generator.world.**WorldEntity**(**args*, ***kwargs*)

Bases: *Variable*

A WorldEntity is an abstract concept representing anything with a name and a type.

Create a Variable.

Parameters

- **name** – The (unique) name of the variable.
- **type** (*optional*) – The type of the variable. Defaults to the same as the name.

add_related_fact(*fact*)

Return type
None

classmethod create(*var*)

Return type
Union[*WorldRoom*, *WorldObject*]

get_attributes()

Return type
List[*Proposition*]

property id: str

Return type
str

name

type

class textworld.generator.world.**WorldObject**(**args*, ***kwargs*)

Bases: *WorldEntity*

A WorldObject is anything we can directly interact with.

Create a Variable.

Parameters

- **name** – The (unique) name of the variable.
- **type** (*optional*) – The type of the variable. Defaults to the same as the name.

name

type

class textworld.generator.world.**WorldRoom**(**args*, ***kwargs*)

Bases: *WorldEntity*

WorldRooms can be linked with each other through exits.

Create a Variable.

Parameters

- **name** – The (unique) name of the variable.
- **type** (*optional*) – The type of the variable. Defaults to the same as the name.

name

type

textworld.generator.world.**connect**(*room1*, *direction*, *room2*, *door=None*)

Generate predicates that connect two rooms.

Parameters

- **room1** (*Variable*) – A room variable.
- **direction** (str) – Direction that we need to travel to go from room1 to room2.
- **room2** (*Variable*) – A room variable.
- **door** (Optional[*Variable*]) – The door separating the two rooms. If None, there is no door between the rooms.

Return type
List[*Proposition*]

textworld.generator.world.**graph2state**(*G*, *rooms*)

Convert Graph object to a list of Proposition.

Parameters

- **G** (Graph) – Graph defining the structure of the world.
- **rooms** (Dict[str, *Variable*]) – information about the rooms in the world.

Return type
List[*Proposition*]

textworld.generator.graph_networks.**create_map**(*rng*, *n_nodes*, *h*, *w*, *possible_door_states=['open', 'closed', 'locked']*)

textworld.generator.graph_networks.**create_small_map**(*rng*, *n_rooms*, *possible_door_states=['open', 'closed', 'locked']*)

textworld.generator.graph_networks.**direction**(*x*, *y*)

textworld.generator.graph_networks.**extremes**(*G*)

Find left most and bottom nodes in the cartesian sense.

textworld.generator.graph_networks.**gen_layout**(*rng*, *n_nodes=5*, *h=10*, *w=10*)

Generate a map with n_nodes rooms by picking a subgraph from a h,w grid.

textworld.generator.graph_networks.**get_path**(*G*, *room1*, *room2*)

textworld.generator.graph_networks.**mark_doors**(*G*, *rng*, *possible_door_states=['open', 'closed', 'locked']*)

Put doors between neighbouring articulation points.

textworld.generator.graph_networks.**plot_graph**(*G*, *show=True*)

Plot TextWorld's graph representation of a world.

Return type
None

textworld.generator.graph_networks.**relabel**(*G*)

Relabel G so that its origin is (0, 0)

textworld.generator.graph_networks.**reverse_direction**(*direction*)

textworld.generator.graph_networks.**shortest_path**(*G*, *source*, *target*)

Return shortest path in terms of directions.

textworld.generator.graph_networks.**xy_diff**(*x*, *y*)

15.3 GameMaker

exception textworld.generator.maker.**ExitAlreadyUsedError**

Bases: ValueError

exception textworld.generator.maker.**FailedConstraintsError**(*failed_constraints*)

Bases: ValueError

Thrown when a constraint has failed during generation.

Parameters
failed_constraints (List[*Action*]) – The constraints that have failed

exception textworld.generator.maker.**MissingPlayerError**

Bases: ValueError

exception textworld.generator.maker.**PlayerAlreadySetError**

Bases: ValueError

exception textworld.generator.maker.**QuestError**

Bases: ValueError

class textworld.generator.maker.**GameMaker**(*options=None*)

Bases: object

Stateful utility class for handcrafting text-based games.

player

Entity representing the player.

Type
WorldEntity

inventory

Entity representing the player's inventory.

Type
WorldEntity

nowhere

List of out-of-world entities (e.g. objects that would only appear later in a game).

Type
List[*WorldEntity*]

rooms

The rooms present in this world.

Type
List[*WorldRoom*]

paths

The connections between the rooms.

Type
List[*WorldPath*]

Creates an empty world, with a player and an empty inventory.

add_fact(*name*, **entities*)

Adds a fact.

Parameters

- **name** (str) – The name of the new fact.
- ***entities** – A list of *WorldEntity* as arguments to this fact.

Return type
None

build(*validate=True*)

Create a Game instance given the defined facts.

Parameters
validate (*optional*) – If True, check if the game is valid, i.e. respects all constraints.

Return type
Generated game.

compile(*path*)

Compile this game.

Parameters
path (str) – Path where to save the generated game.

Returns
Path to the game file.

Return type
game_file

connect(*exit1*, *exit2*)

Connect two rooms using their exits.

Parameters

- **exit1** (*WorldRoomExit*) – The exit of the first room to link.
- **exit2** (*WorldRoomExit*) – The exit of the second room to link.

Return type
WorldPath

Returns
The path created by the link between two rooms, with no door.

find_by_name(*name*)

Find an entity using its name.

Return type
Optional[*WorldEntity*]

find_path(*room1*, *room2*)

Get the path between two rooms, if it exists.

Parameters

- **room1** (*WorldRoom*) – One of the two rooms.
- **room2** (*WorldRoom*) – The other room.

Return type
Optional[*WorldEntity*]

Returns
The matching path path, if it exists.

findall(*type*)

Gets all entities of the given type.

Parameters
type (str) – The type of entity to find.

Return type
List[*WorldEntity*]

Returns
All entities which match.

generate_distractors(*nb_distractors*)

Generates a number of distractors - random objects.

Parameters
nb_distractors (int) – The number of distractors to game will contain.

Return type
None

generate_random_quests(*nb_quests=1*, *length=1*, *breadth=1*)

Generates random quests for the game.

> **Warning:** This method overrides any previous quests the game had.

Parameters

- **nb_quests** – Number of parallel quests, i.e. not sharing a common goal.
- **length** (int) – Number of actions that need to be performed to complete the game.
- **breadth** (int) – Number of subquests per independent quest. It controls how nonlinear a quest can be (1: linear).

Return type
List[*Quest*]

Returns
The generated quests.

import_graph(*G*)

Convert Graph object to a list of Proposition.

Parameters
G (Graph) – Graph defining the structure of the world.

Return type
List[*WorldRoom*]

move(*entity*, *new_location*)

Move an entity to a new location.

Parameters

- **entity** (*WorldEntity*) – Entity to move.
- **new_location** (*WorldEntity*) – Where to move the entity.

Return type
None

new(*type*, *name=None*, *desc=None*)

Creates new entity given its type.

Parameters

- **type** (str) – The type of the entity.
- **name** (Optional[str]) – The name of the entity.
- **desc** (Optional[str]) – The description of the entity.

Return type
Union[*WorldEntity*, *WorldRoom*]

Returns

The newly created entity.

- If the type is 'r', then a *WorldRoom* object is returned.
- Otherwise, a *WorldEntity* is returned.

new_door(*path*, *name=None*, *desc=None*)

Creates a new door and add it to the path.

Parameters

- **path** (*WorldPath*) – A path between two rooms where to add the door.
- **name** (Optional[str]) – The name of the door. Default: generate one automatically.
- **desc** (Optional[str]) – The description of the door.

Return type
WorldEntity

Returns
The newly created door.

new_event_using_commands(*commands*)

Creates a new event using predefined text commands.

This launches a textworld.play session to execute provided commands.

Parameters
commands (List[str]) – Text commands.

Return type
Event

Returns
The resulting event.

new_fact(*name*, **entities*)

Create new fact.

Parameters

- **name** (str) – The name of the new fact.
- ***entities** – A list of entities as arguments to the new fact.

Return type
None

new_quest_using_commands(*commands*)

Creates a new quest using predefined text commands.

This launches a `textworld.play` session to execute provided commands.

Parameters
commands (List[str]) – Text commands.

Return type
Quest

Returns
The resulting quest.

new_room(*name=None*, *desc=None*)

Create new room entity.

Parameters

- **name** (Optional[str]) – The name of the room.
- **desc** (Optional[str]) – The description of the room.

Return type
WorldRoom

Returns
The newly created room entity.

record_quest()

Defines the game's quest by recording the commands.

This launches a `textworld.play` session.

Return type
Quest

Returns
The resulting quest.

render(*interactive=False*)

Returns a visual representation of the world. :type interactive: bool :param interactive: opens an interactive session in the browser instead of returning a png. :return: :param save_screenshot: ONLY FOR WHEN interactive == False. Save screenshot in temp directory. :param filename: filename for screenshot

set_player(*room*)

Place the player in room.

Parameters
room (*WorldRoom*) – The room the player will start in.

Notes

At the moment, the player can only be place once and cannot be moved once placed.

Raises
PlayerAlreadySetError – If the player has already been set.

Return type
None

set_quest_from_commands(*commands*)

Defines the game's quest using predefined text commands.

This launches a `textworld.play` session.

Parameters
commands (List[str]) – Text commands.

Return type
Quest

Returns
The resulting quest.

set_walkthrough(*commands*)

test(*walkthrough=False*)

Test the game being built.

This launches a `textworld.play` session.

Return type
None

validate()

Check if the world is valid and can be compiled.

A world is valid is the player has been place in a room and all constraints (defined in the *knowledge base*) are respected.

Return type
bool

property facts: Iterable[*Proposition*]

All the facts associated to the current game state.

Return type
Iterable[*Proposition*]

property state: *State*

Current state of the world.

Return type
State

class textworld.generator.maker.**WorldEntity**(*var*, *name=None*, *desc=None*, *kb=None*)

Bases: object

Represents an entity in the world.

Example of entities commonly found in text-based games: rooms, doors, items, etc.

Parameters

- **var** (*Variable*) – The underlying variable for the entity which is used by TextWorld's inference engine.
- **name** (Optional[str]) – The name of the entity that will be displayed in-game. Default: generate one according the variable's type.
- **desc** (Optional[str]) – The description of the entity that will be displayed when examining it in the game.

add(**entities*)

Add children to this entity.

Return type
None

add_fact(*name*, **entities*)

Adds a fact to this entity.

Parameters

- **name** (str) – The name of the new fact.
- ***entities** – A list of entities as arguments to the new fact.

Return type
None

add_property(*name*)

Adds a property to this entity.

A property is a fact that only involves one entity. For instance, 'closed(c)', 'open(c)', and 'locked(c)' are all properties.

Parameters
name (str) – The name of the new property.

Return type
None

has_property(*name*)

Determines if this object has a property with the given name.

Parameters
property. (*The name of the*) –

Example

```
>>> from textworld import GameMaker
>>> M = GameMaker()
>>> chest = M.new(type="c", name="chest")
>>> chest.has_property('closed')
False
>>> chest.add_property('closed')
>>> chest.has_property('closed')
True
```

Return type
bool

remove(**entities*)

remove_fact(*name*, **entities*)

Return type
None

remove_property(*name*)

Return type
None

property facts: List[*Proposition*]

All facts related to this entity (or its children content).

Return type
List[*Proposition*]

property id: str

Unique name used internally.

Return type
str

property name: str

Name of this entity.

Return type
str

property properties: List[*Proposition*]

Properties of this object are things that refer to this object and this object alone. For instance, 'closed', 'open', and 'locked' are possible properties of 'containers'.

Return type
List[*Proposition*]

property type: str

Type of this entity.

Return type
str

class textworld.generator.maker.**WorldPath**(*src*, *src_exit*, *dest*, *dest_exit*, *door=None*, *kb=None*)

Bases: object

Represents a path between two *WorldRoom* objects.

A *WorldPath* encapsulates the source *WorldRoom*, the source *WorldRoomExit*, the destination *WorldRoom* and the destination *WorldRoom*. Optionally, a linking door can also be provided.

Parameters

- **src** (*WorldRoom*) – The source room.
- **src_exit** (*WorldRoomExit*) – The exit of the source room.
- **dest** (*WorldRoom*) – The destination room.
- **dest_exit** (*WorldRoomExit*) – The exist of the destination room.
- **door** (Optional[*WorldEntity*]) – The door between the two rooms, if any.

property door: Optional[*WorldEntity*]

The entity representing the door or None if there is none.

Return type
Optional[*WorldEntity*]

property facts: List[*Proposition*]

Facts related to this path.

Return type
List[*Proposition*]

Returns
The facts that make up this path.

class textworld.generator.maker.**WorldRoom**(**args*, ***kwargs*)

Bases: *WorldEntity*

Represents a room in the world.

Takes the same arguments as WorldEntity.

Then, creates a WorldRoomExit for each direction defined in graph_networks.DIRECTIONS, and sets exits to be a dict of those names to the newly created rooms. It then sets an attribute to each name.

Parameters

- **args** – The args to pass to WorldEntity
- **kwargs** – The kwargs to pass to WorldEntity

east

north

south

west

class textworld.generator.maker.**WorldRoomExit**(*src*, *direction*, *dest=None*)

Bases: object

Represents an exit from a Room.

These are used to connect `WorldRoom`s to form `WorldPath`s. `WorldRoomExit`s are linked to each other through their :py:attr:`dest`.

When dest is None, it means there is no path leading to this exit yet.

Parameters

- **src** (*WorldRoom*) – The WorldRoom that the exit is from.
- **direction** (str) – The direction the exit is in: north, east, south, and west are common.
- **dest** (Optional[*WorldRoom*]) – The WorldRoomExit that this exit links to (exits are linked to each other).

textworld.generator.maker.**get_failing_constraints**(*state*, *kb=None*)

15.4 Grammar

class textworld.generator.text_generation.**CountOrderedDict**

Bases: OrderedDict

An OrderedDict whose empty items are 0

class textworld.generator.text_generation.**MergeAction**

Bases: object

Group of actions merged into one.

This allows for blending consecutive instructions.

textworld.generator.text_generation.**assign_description_to_object**(*obj*, *grammar*, *game*)

Assign a descripton to an object.

textworld.generator.text_generation.**assign_description_to_quest**(*quest*, *game*, *grammar*)

textworld.generator.text_generation.**assign_description_to_room**(*room*, *game*, *grammar*)

Assign a descripton to a room.

textworld.generator.text_generation.**assign_name_to_object**(*obj*, *grammar*, *game_infos*)

Assign a name to an object (if needed).

textworld.generator.text_generation.**assign_new_matching_names**(*obj1_infos*, *obj2_infos*, *grammar*, *exclude*)

textworld.generator.text_generation.**clean_replace_objs**(*grammar*, *desc*, *objs*, *game*)

Return a cleaned/keyword replaced for a list of objects.

textworld.generator.text_generation.**describe_event**(*event*, *game*, *grammar*)

Assign a descripton to a quest.

Return type
str

textworld.generator.text_generation.**expand_clean_replace**(*symbol*, *grammar*, *obj*, *game*)

Return a cleaned/keyword replaced symbol.

textworld.generator.text_generation.**generate_instruction**(*action*, *grammar*, *game*, *counts*)

Generate text instruction for a specific action.

textworld.generator.text_generation.**generate_text_from_grammar**(*game*, *grammar*)

textworld.generator.text_generation.**get_action_chains**(*actions*, *grammar*, *game*)

Reduce the action list by combining similar actions.

textworld.generator.text_generation.**is_seq**(*chain*, *game*)

Check if we have a theoretical chain in actions.

textworld.generator.text_generation.**list_to_string**(*lst*, *det*, *det_type='a'*)

Convert a list to a natural language string.

textworld.generator.text_generation.**obj_list_to_prop_string**(*objs*, *property*, *game*, *det=True*, *det_type='a'*)

Convert an object list to a nl string list of names.

textworld.generator.text_generation.**repl_sing_plur**(*phrase*, *length*)

Alter a sentence depending on whether or not we are dealing with plural or singular objects (for counting)

textworld.generator.text_generation.**replace_num**(*phrase*, *val*)

Add a numerical value to a string.

exception textworld.generator.text_grammar.**MissingTextGrammar**(*path*)

Bases: NameError

class textworld.generator.text_grammar.**Grammar**(*options={}*, *rng=None*, *kb=None*)

Bases: object

Context-Free Grammar for text generation.

Parameters

- **options** (Union[*GrammarOptions*, Mapping[str, Any]]) – For customizing text generation process (see *textworld.generator.GrammarOptions* for the list of available options).
- **rng** (Optional[RandomState]) – Random generator used for sampling tag expansions.

check()

Check if this grammar is valid.

TODO: use logging mechanism to report warnings and errors.

Return type
bool

expand(*text*, *rng=None*)

Expand some text until there is no more tag to expand.

Parameters

- **text** (str) – Text potentially containing grammar tags to be expanded.
- **rng** (*optional*) – Random generator used to chose an expansion when there is many. By default, it used the random generator of this grammar object.

Returns
Resulting text in which there is no grammar tag left to be expanded.

Return type
expanded_text

generate_name(*obj_type*, *room_type=''*, *include_adj=None*, *exclude=[]*)

Generate a name given an object type and the type room it belongs to.

Parameters

- **obj_type** (str) – Type of the object for which we will generate a name.
- **room_type** (*optional*) – Type of the room the object belongs to.
- **include_adj** (*optional*) – If True, the name can contain a generated adjective. If False, any generated adjective will be discarded. Default: use value grammar.options.include_adj
- **exclude** (*optional*) – List of names we should avoid generating.

Return type
Tuple[str, str, str]

Returns

- *name* – The whole name, i.e. `adj + " " + noun`.
- *adj* – The adjective part of the name.
- *noun* – The noun part of the name.

`get_all_adjective_for_type`(*type*)

Get all possible adjectives for a given object type.

Parameters
type (`str`) – Object type.

Returns
All possible adjectives sorted in alphabetical order.

Return type
adjectives

`get_all_expansions_for_tag`(*tag*, *max_depth=500*)

Get all possible expansions for a grammar tag.

Parameters

- **tag** (`str`) – Grammar tag to be expanded.
- **max_depth** (*optional*) – Maximum recursion depth when expanding tag.

Returns
All possible expansions.

Return type
expansions

`get_all_expansions_for_type`(*type*)

Get all possible expansions for a given object type.

Parameters
type (`str`) – Object type.

Returns
All possible names.

Return type
names

`get_all_names_for_type`(*type*, *include_adj*)

Get all possible names for a given object type.

Parameters

- **type** (`str`) – Object type.
- **include_adj** (*optional*) – If True, names can contain generated adjectives. If False, any generated adjectives will be discarded.

Returns
All possible names sorted in alphabetical order.

Return type
names

`get_all_nouns_for_type`(*type*)

Get all possible nouns for a given object type.

Parameters
type (str) – Object type.

Returns
All possible nouns sorted in alphabetical order.

Return type
nouns

get_random_expansion(*tag*, *rng=None*)

Return a randomly chosen expansion for the given tag.

Parameters

- **tag** (str) – Grammar tag to be expanded.
- **rng** (*optional*) – Random generator used to chose an expansion when there is many. By default, it used the random generator of this grammar object.

Returns
An expansion chosen randomly for the provided tag.

Return type
expansion

get_vocabulary()

Return type
List[str]

has_tag(*tag*)

Check if the grammar has a given tag.

Return type
bool

split_name_adj_noun(*candidate*, *include_adj*)

Extract the full name, the adjective and the noun from a string.

Parameters

- **candidate** (str) – String that may contain one adjective-noun sperator '|'.
- **include_adj** (*optional*) – If True, the name can contain a generated adjective. If False, any generated adjective will be discarded.

Return type
Optional[Tuple[str, str, str]]

Returns

- *name* – The whole name, i.e. adj + " " + noun.
- *adj* – The adjective part of the name.
- *noun* – The noun part of the name.

class textworld.generator.text_grammar.**GrammarOptions**(*options=None*, ***kwargs*)

Bases: object

copy()

Return type
GrammarOptions

classmethod deserialize(*data*)

Creates a *GrammarOptions* from serialized data.

Parameters
data (Mapping) – Serialized data with the needed information to build a *GrammarOptions* object.

Return type
GrammarOptions

serialize()

Serialize this object.

Results:
GrammarOptions's data serialized to be JSON compatible.

Return type
Mapping

allowed_variables_numbering

Append numbers after an object name if there is not enough variation for it.

Type
bool

ambiguous_instructions

When True, in the game objective, objects of interest might be refer to by their type or adjective rather than full name.

Type
bool

blend_descriptions

When True, objects sharing some properties might be described in a single sentence rather than separate consecutive ones.

Type
bool

blend_instructions

When True, consecutive actions to be accomplished might be described in a single sentence rather than separate ones.

Type
bool

include_adj

When True, object names can be preceeded by an adjective.

Type
bool

names_to_exclude

List of names the text generation should not use.

Type
List[str]

only_last_action

When True, only the last action of a quest will be described in the generated objective.

Type
bool

theme

Grammar theme's name. All `*.twg` files starting with that name will be loaded.

Type
str

unique_expansion

When True, #symbol# are force to be expanded to unique text.

Type
bool

property uuid: str

Generate UUID for this set of grammar options.

Return type
str

textworld.generator.text_grammar.**fix_determinant**(*var*)

15.5 Knowledge Base

class textworld.generator.data.**KnowledgeBase**(*logic*, *text_grammars_path*)

Bases: object

classmethod default()

classmethod deserialize(*data*)

Return type
KnowledgeBase

get_reverse_action(*action*)

classmethod load(*target_dir=None*, *logic_path=None*, *grammar_path=None*)

Build a KnowledgeBase from several files (logic and text grammar).

Parameters

- **target_dir** (Optional[str]) – Folder containing two subfolders: `logic` and `text_grammars`. If provided, both `logic_path` and `grammar_path` are ignored.
- **logic_path** (Optional[str]) – Folder containing `*.twl` files that describe the logic of a game.
- **grammar_path** (Optional[str]) – Folder containing `*.twg` files that describe the grammar used for text generation.

Return type
KnowledgeBase

Returns
KnowledgeBase object.

serialize()

Return type
str

textworld.generator.data.**create_data_files**(*dest='./textworld_data'*, *verbose=False*, *force=False*)

Creates grammar files in the target directory.

Will NOT overwrite files if they alredy exist (checked on per-file basis).

Parameters

- **dest** (str) – The path to the directory where to dump the data files into.
- **verbose** (bool) – Print when skipping an existing file.
- **force** (bool) – Overwrite all existing files.

15.5.1 Data

container.twl

```
# container
type c : t {
    predicates {
        open(c);
        closed(c);
        locked(c);

        in(o, c);
    }

    rules {
        lock/c   :: $at(P, r) & $at(c, r) & $in(k, I) & $match(k, c) & closed(c) ->␣
↪locked(c);
        unlock/c :: $at(P, r) & $at(c, r) & $in(k, I) & $match(k, c) & locked(c) ->␣
↪closed(c);

        open/c  :: $at(P, r) & $at(c, r) & closed(c) -> open(c);
        close/c :: $at(P, r) & $at(c, r) & open(c) -> closed(c);
    }

    reverse_rules {
        lock/c :: unlock/c;
        open/c :: close/c;
    }

    constraints {
        c1 :: open(c)   & closed(c) -> fail();
        c2 :: open(c)   & locked(c) -> fail();
        c3 :: closed(c) & locked(c) -> fail();
    }

    inform7 {
        type {
```

(continues on next page)

(continued from previous page)

```
            kind :: "container";
            definition :: "containers are openable, lockable and fixed in place.␣
↪containers are usually closed.";
        }

        predicates {
            open(c) :: "The {c} is open";
            closed(c) :: "The {c} is closed";
            locked(c) :: "The {c} is locked";

            in(o, c) :: "The {o} is in the {c}";
        }

        commands {
            open/c :: "open {c}" :: "opening the {c}";
            close/c :: "close {c}" :: "closing the {c}";

            lock/c :: "lock {c} with {k}" :: "locking the {c} with the {k}";
            unlock/c :: "unlock {c} with {k}" :: "unlocking the {c} with the {k}";
        }
    }
}
```

door.twl

```
# door
type d : t {
    predicates {
        open(d);
        closed(d);
        locked(d);

        link(r, d, r);
    }

    rules {
        lock/d   :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & $in(k, I) & $match(k,
↪ d) & closed(d) -> locked(d);
        unlock/d :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & $in(k, I) & $match(k,
↪ d) & locked(d) -> closed(d);

        open/d   :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & closed(d) -> open(d)␣
↪& free(r, r') & free(r', r);
        close/d  :: $at(P, r) & $link(r, d, r') & $link(r', d, r) & open(d) & free(r, r
↪') & free(r', r) -> closed(d);

        examine/d :: at(P, r) & $link(r, d, r') -> at(P, r);  # Nothing changes.
    }

    reverse_rules {
```

(continues on next page)

(continued from previous page)

```
        lock/d :: unlock/d;
        open/d :: close/d;

        examine/d :: examine/d;
    }

    constraints {
        d1 :: open(d)   & closed(d) -> fail();
        d2 :: open(d)   & locked(d) -> fail();
        d3 :: closed(d) & locked(d) -> fail();

        # A door can't be used to link more than two rooms.
        link1 :: link(r, d, r') & link(r, d, r'') -> fail();
        link2 :: link(r, d, r') & link(r'', d, r''') -> fail();

        # There's already a door linking two rooms.
        link3 :: link(r, d, r') & link(r, d', r') -> fail();

        # There cannot be more than four doors in a room.
        too_many_doors :: link(r, d1: d, r1: r) & link(r, d2: d, r2: r) & link(r, d3: d,␣
↪r3: r) & link(r, d4: d, r4: r) & link(r, d5: d, r5: r) -> fail();

        # There cannot be more than four doors in a room.
        dr1 :: free(r, r1: r) & link(r, d2: d, r2: r) & link(r, d3: d, r3: r) & link(r,␣
↪d4: d, r4: r) & link(r, d5: d, r5: r) -> fail();
        dr2 :: free(r, r1: r) & free(r, r2: r) & link(r, d3: d, r3: r) & link(r, d4: d,␣
↪r4: r) & link(r, d5: d, r5: r) -> fail();
        dr3 :: free(r, r1: r) & free(r, r2: r) & free(r, r3: r) & link(r, d4: d, r4: r) &
↪ link(r, d5: d, r5: r) -> fail();
        dr4 :: free(r, r1: r) & free(r, r2: r) & free(r, r3: r) & free(r, r4: r) &␣
↪link(r, d5: d, r5: r) -> fail();

        free1 :: link(r, d, r') & free(r, r') & closed(d) -> fail();
        free2 :: link(r, d, r') & free(r, r') & locked(d) -> fail();
    }

    inform7 {
        type {
            kind :: "door";
            definition :: "door is openable and lockable.";
        }

        predicates {
            open(d) :: "The {d} is open";
            closed(d) :: "The {d} is closed";
            locked(d) :: "The {d} is locked";
            link(r, d, r') :: "";  # No equivalent in Inform7.
        }

        commands {
            open/d :: "open {d}" :: "opening {d}";
            close/d :: "close {d}" :: "closing {d}";
```

(continues on next page)

(continued from previous page)

```
            unlock/d :: "unlock {d} with {k}" :: "unlocking {d} with the {k}";
            lock/d :: "lock {d} with {k}" :: "locking {d} with the {k}";

            examine/d :: "examine {d}" :: "examining {d}";
        }
    }
}
```

food.twl

```
# food
type f : o {
    predicates {
        edible(f);
        eaten(f);
    }

    rules {
        eat :: in(f, I) -> eaten(f);
    }

    constraints {
        eaten1 :: eaten(f) & in(f, I) -> fail();
        eaten2 :: eaten(f) & in(f, c) -> fail();
        eaten3 :: eaten(f) & on(f, s) -> fail();
        eaten4 :: eaten(f) & at(f, r) -> fail();
    }

    inform7 {
        type {
            kind :: "food";
            definition :: "food is edible.";
        }

        predicates {
            edible(f) :: "The {f} is edible";
            eaten(f) :: "The {f} is nowhere";
        }

        commands {
            eat :: "eat {f}" :: "eating the {f}";
        }
    }
}
```

inventory.twl

```
# Inventory
type I {
    predicates {
        in(o, I);
    }

    rules {
        inventory :: at(P, r) -> at(P, r);  # Nothing changes.

        take :: $at(P, r) & at(o, r) -> in(o, I);
        drop :: $at(P, r) & in(o, I) -> at(o, r);

        take/c :: $at(P, r) & $at(c, r) & $open(c) & in(o, c) -> in(o, I);
        insert :: $at(P, r) & $at(c, r) & $open(c) & in(o, I) -> in(o, c);

        take/s :: $at(P, r) & $at(s, r) & on(o, s) -> in(o, I);
        put    :: $at(P, r) & $at(s, r) & in(o, I) -> on(o, s);

        examine/I :: in(o, I) -> in(o, I);  # Nothing changes.
        examine/s :: at(P, r) & $at(s, r) & $on(o, s) -> at(P, r);  # Nothing changes.
        examine/c :: at(P, r) & $at(c, r) & $open(c) & $in(o, c) -> at(P, r);  # Nothing␣
↪changes.
    }

    reverse_rules {
        inventory :: inventory;

        take :: drop;
        take/c :: insert;
        take/s :: put;

        examine/I :: examine/I;
        examine/s :: examine/s;
        examine/c :: examine/c;
    }

    inform7 {
        predicates {
            in(o, I) :: "The player carries the {o}";
        }

        commands {
            take :: "take {o}" :: "taking the {o}";
            drop :: "drop {o}" :: "dropping the {o}";

            take/c :: "take {o} from {c}" :: "removing the {o} from the {c}";
            insert :: "insert {o} into {c}" :: "inserting the {o} into the {c}";

            take/s :: "take {o} from {s}" :: "removing the {o} from the {s}";
            put :: "put {o} on {s}" :: "putting the {o} on the {s}";
```

(continues on next page)

(continued from previous page)

```
            inventory :: "inventory" :: "taking inventory";

            examine/I :: "examine {o}" :: "examining the {o}";
            examine/s :: "examine {o}" :: "examining the {o}";
            examine/c :: "examine {o}" :: "examining the {o}";
        }
    }
}
```

key.twl

```
# key
type k : o {
    predicates {
        match(k, c);
        match(k, d);
    }

    constraints {
        k1 :: match(k, c) & match(k', c) -> fail();
        k2 :: match(k, c) & match(k, c') -> fail();
        k3 :: match(k, d) & match(k', d) -> fail();
        k4 :: match(k, d) & match(k, d') -> fail();
    }

    inform7 {
        type {
            kind :: "key";
        }

        predicates {
            match(k, c) :: "The matching key of the {c} is the {k}";
            match(k, d) :: "The matching key of the {d} is the {k}";
        }
    }
}
```

object.twl

```
# object
type o : t {
    constraints {
        obj1 :: in(o, I) & in(o, c) -> fail();
        obj2 :: in(o, I) & on(o, s) -> fail();
        obj3 :: in(o, I) & at(o, r) -> fail();
        obj4 :: in(o, c) & on(o, s) -> fail();
        obj5 :: in(o, c) & at(o, r) -> fail();
        obj6 :: on(o, s) & at(o, r) -> fail();
        obj7 :: at(o, r) & at(o, r') -> fail();
```

(continues on next page)

(continued from previous page)

```
        obj8 :: in(o, c) & in(o, c') -> fail();
        obj9 :: on(o, s) & on(o, s') -> fail();
    }

    inform7 {
        type {
            kind :: "object-like";
            definition :: "object-like is portable.";
        }
    }
}
```

player.twl

```
# Player
type P {
    rules {
        look :: at(P, r) -> at(P, r);  # Nothing changes.
    }

    reverse_rules {
        look :: look;
    }

    inform7 {
        commands {
            look :: "look" :: "looking";
        }
    }
}
```

room.twl

```
# room
type r {
    predicates {
        at(P, r);
        at(t, r);

        north_of(r, r);
        west_of(r, r);

        north_of/d(r, d, r);
        west_of/d(r, d, r);

        free(r, r);

        south_of(r, r') = north_of(r', r);
        east_of(r, r') = west_of(r', r);
```

(continues on next page)

(continued from previous page)

```
        south_of/d(r, d, r') = north_of/d(r', d, r);
        east_of/d(r, d, r') = west_of/d(r', d, r);
    }

    rules {
        go/north :: at(P, r) & $north_of(r', r) & $free(r, r') & $free(r', r) -> at(P, r
↪');
        go/south :: at(P, r) & $south_of(r', r) & $free(r, r') & $free(r', r) -> at(P, r
↪');
        go/east  :: at(P, r) & $east_of(r', r) & $free(r, r') & $free(r', r) -> at(P, r
↪');
        go/west  :: at(P, r) & $west_of(r', r) & $free(r, r') & $free(r', r) -> at(P, r
↪');
    }

    reverse_rules {
        go/north :: go/south;
        go/west :: go/east;
    }

    constraints {
        r1 :: at(P, r) & at(P, r') -> fail();
        r2 :: at(s, r) & at(s, r') -> fail();
        r3 :: at(c, r) & at(c, r') -> fail();

        # An exit direction can only lead to one room.
        nav_rr1 :: north_of(r, r') & north_of(r'', r') -> fail();
        nav_rr2 :: south_of(r, r') & south_of(r'', r') -> fail();
        nav_rr3 :: east_of(r, r') & east_of(r'', r') -> fail();
        nav_rr4 :: west_of(r, r') & west_of(r'', r') -> fail();

        # Two rooms can only be connected once with each other.
        nav_rrA :: north_of(r, r') & south_of(r, r') -> fail();
        nav_rrB :: north_of(r, r') & west_of(r, r') -> fail();
        nav_rrC :: north_of(r, r') & east_of(r, r') -> fail();
        nav_rrD :: south_of(r, r') & west_of(r, r') -> fail();
        nav_rrE :: south_of(r, r') & east_of(r, r') -> fail();
        nav_rrF :: west_of(r, r')  & east_of(r, r') -> fail();
    }

    inform7 {
        type {
            kind :: "room";
        }

        predicates {
            at(P, r) :: "The player is in {r}";
            at(t, r) :: "The {t} is in {r}";
            free(r, r') :: "";  # No equivalent in Inform7.

            north_of(r, r') :: "The {r} is mapped north of {r'}";
```

(continues on next page)

(continued from previous page)

```
            south_of(r, r') :: "The {r} is mapped south of {r'}";
            east_of(r, r') :: "The {r} is mapped east of {r'}";
            west_of(r, r') :: "The {r} is mapped west of {r'}";

            north_of/d(r, d, r') :: "South of {r} and north of {r'} is a door called {d}
↪";
            south_of/d(r, d, r') :: "North of {r} and south of {r'} is a door called {d}
↪";
            east_of/d(r, d, r') :: "West of {r} and east of {r'} is a door called {d}";
            west_of/d(r, d, r') :: "East of {r} and west of {r'} is a door called {d}";
        }

        commands {
            go/north :: "go north" :: "going north";
            go/south :: "go south" :: "going south";
            go/east :: "go east" :: "going east";
            go/west :: "go west" :: "going west";
        }
    }
}
```

supporter.twl

```
# supporter
type s : t {
    predicates {
        on(o, s);
    }

    inform7 {
        type {
            kind :: "supporter";
            definition :: "supporters are fixed in place.";
        }

        predicates {
            on(o, s) :: "The {o} is on the {s}";
        }
    }
}
```

thing.twl

```
# thing
type t {
    rules {
        examine/t :: at(P, r) & $at(t, r) -> at(P, r);
    }

    reverse_rules {
        examine/t :: examine/t;
    }

    inform7 {
        type {
            kind :: "thing";
        }

        commands {
            examine/t :: "examine {t}" :: "examining the {t}";
        }
    }
}
```

15.6 Inform 7

exception textworld.generator.inform7.world2inform7.**CouldNotCompileGameError**

Bases: RuntimeError

exception textworld.generator.inform7.world2inform7.**TextworldInform7Warning**

Bases: UserWarning

class textworld.generator.inform7.world2inform7.**Inform7Game**(*game*)

Bases: object

define_inform7_kinds()

Generate Inform 7 kind definitions.

Return type
str

detect_action(*i7_event*, *actions*)

Detect which action corresponds to a Inform7 event.

Parameters

- **i7_event** (str) – Inform7 event detected.
- **actions** (Iterable[*Action*]) – List of action to match the Inform7 event against.

Return type
Optional[*Action*]

Returns
Action corresponding to the provided Inform7 event.

gen_commands_from_actions(*actions*)

Return type
List[str]

gen_source(*seed=1234*)

Return type
str

gen_source_for_attribute(*attr*)

Return type
Optional[str]

gen_source_for_attributes(*attributes*)

Return type
str

gen_source_for_conditions(*conds*)

Generate Inform 7 source for winning/losing conditions.

Return type
str

gen_source_for_map(*src_room*)

Return type
str

gen_source_for_objects(*objects*)

Return type
str

gen_source_for_rooms()

Return type
str

get_human_readable_action(*action*)

Return type
Action

get_human_readable_fact(*fact*)

Return type
Proposition

VERSION = 1

textworld.generator.inform7.world2inform7.**compile_inform7_game**(*source*, *output*, *verbose=False*)

Return type
None

textworld.generator.inform7.world2inform7.**generate_inform7_source**(*game*, *seed=1234*, *use_i7_description=False*)

Return type
str

textworld.generator.inform7.world2inform7.**split_string**(*string*, *name*, *cutoff=200*)

CHAPTER

SIXTEEN

TEXTWORLD.CHALLENGES

CHAPTER

SEVENTEEN

TEXTWORLD.LOGIC

class textworld.logic.**Action**(*name*, *preconditions*, *postconditions*)

Bases: object

An action in the environment.

Create an Action.

Parameters

- **name** (str) – The name of this action.
- **preconditions** (Iterable[*Proposition*]) – The preconditions that must hold before this action is applied.
- **postconditions** (Iterable[*Proposition*]) – The conditions that replace the preconditions once applied.

classmethod deserialize(*data*)

Return type
Action

format_command(*mapping={}*)

inverse(*name=None*)

Invert the direction of this action.

Parameters
name (*optional*) – The new name for the inverse action.

Return type
An action that does the exact opposite of this one.

classmethod parse(*expr*)

Parse an action expression.

Parameters
expr (str) – The string to parse, in the form `name :: [$]proposition [& [$]proposition]* -> proposition [& proposition]*`.

Return type
Action

serialize()

Return type
Mapping

property added: Collection[*Proposition*]

All the new propositions being introduced by this action.

Return type
Collection[*Proposition*]

property all_propositions: Collection[*Proposition*]

All the pre- and post-conditions.

Return type
Collection[*Proposition*]

property removed: Collection[*Proposition*]

All the old propositions being removed by this action.

Return type
Collection[*Proposition*]

property variables

class textworld.logic.**Alias**(*pattern*, *replacement*)

Bases: object

A shorthand predicate alias.

expand(*predicate*)

Expand a use of this alias into its replacement.

Return type
Collection[*Predicate*]

class textworld.logic.**GameLogic**

Bases: object

The logic for a game (types, rules, etc.).

classmethod deserialize(*data*)

Return type
GameLogic

classmethod load(*paths*)

normalize_rule(*rule*)

Return type
Rule

classmethod parse(*cls*, *document*)

Return type
GameLogic

serialize()

Return type
str

class textworld.logic.**Inform7Command**(*rule*, *command*, *event*)

Bases: object

Information about an Inform 7 command.

class textworld.logic.**Inform7Logic**

Bases: object

The Inform 7 bindings of a GameLogic.

class textworld.logic.**Inform7Predicate**(*predicate*, *source*)

Bases: object

Information about an Inform 7 predicate.

class textworld.logic.**Inform7Type**(*name*, *kind*, *definition=None*)

Bases: object

Information about an Inform 7 kind.

class textworld.logic.**Placeholder**(*name*, *type=None*)

Bases: object

A symbolic placeholder for a variable in a Predicate.

Create a Placeholder.

Parameters

- **name** (str) – The name of this placeholder.
- **type** (*optional*) – The type of variable represented. Defaults to the name with any trailing apostrophes stripped.

classmethod deserialize(*data*)

Return type
Placeholder

classmethod parse(*expr*)

Parse a placeholder expression.

Parameters
expr (str) – The string to parse, in the form *name* or name: type.

Return type
Placeholder

serialize()

Return type
Mapping

name

type

class textworld.logic.**Predicate**(*name*, *parameters*)

Bases: object

A boolean-valued function over variables.

Create a Predicate.

Parameters

- **name** (str) – The name of this predicate.
- **parameters** (Iterable[*Placeholder*]) – The symbolic arguments to this predicate.

classmethod deserialize(*data*)

Return type
Predicate

instantiate(*mapping*)

Instantiate this predicate with the given mapping.

Parameters
mapping (Mapping[*Placeholder*, *Variable*]) – A mapping from Placeholders to Variables.

Return type
The instantiated Proposition with each Placeholder mapped to the corresponding Variable.

match(*proposition*)

Match this predicate against a concrete proposition.

Parameters
proposition (*Proposition*) – The proposition to match against.

Return type
Optional[Mapping[*Placeholder*, *Variable*]]

Returns

- The mapping from placeholders to variables such that `self.instantiate(mapping) == proposition`, or None if no
- *such mapping exists.*

classmethod parse(*expr*)

Parse a predicate expression.

Parameters
expr (str) – The string to parse, in the form `name(placeholder [, placeholder]*)`.

Return type
Predicate

serialize()

Return type
Mapping

substitute(*mapping*)

Copy this predicate, substituting certain placeholders for others.

Parameters
mapping (Mapping[*Placeholder*, *Placeholder*]) – A mapping from old to new placeholders.

Return type
Predicate

property names: Collection[str]

The names of the placeholders in this predicate.

Return type
Collection[str]

property types: Collection[str]

The types of the placeholders in this predicate.

Return type
Collection[str]

class textworld.logic.**Proposition**(*name*, *arguments=[]*)

Bases: NewBase

An instantiated Predicate, with concrete variables for each placeholder.

Create a Proposition.

Parameters

- **name** (str) – The name of the proposition.
- **arguments** (Iterable[*Variable*]) – The variables this proposition is applied to.

classmethod deserialize(*data*)

Return type
Proposition

classmethod parse(*expr*)

Parse a proposition expression.

Parameters
expr (str) – The string to parse, in the form `name(variable [, variable]*)`.

Return type
Proposition

serialize()

Return type
Mapping

arguments

name

property names: Collection[str]

The names of the variables in this proposition.

Return type
Collection[str]

signature

property types: Collection[str]

The types of the variables in this proposition.

Return type
Collection[str]

class textworld.logic.**Rule**(*name*, *preconditions*, *postconditions*)

Bases: object

A template for an action.

Create a Rule.

Parameters

- **name** (str) – The name of this rule.
- **preconditions** (Iterable[*Predicate*]) – The preconditions that must hold before this rule is applied.
- **postconditions** (Iterable[*Predicate*]) – The conditions that replace the preconditions once applied.

classmethod deserialize(*data*)

Return type
Rule

instantiate(*mapping*)

Instantiate this rule with the given mapping.

Parameters
mapping (Mapping[*Placeholder*, *Variable*]) – A mapping from Placeholders to Variables.

Return type
The instantiated Action with each Placeholder mapped to the corresponding Variable.

inverse(*name=None*)

Invert the direction of this rule.

Parameters
name (*optional*) – The new name for the inverse rule.

Return type
A rule that does the exact opposite of this one.

match(*action*)

Match this rule against a concrete action.

Parameters
action (*Action*) – The action to match against.

Return type
Optional[Mapping[*Placeholder*, *Variable*]]

Returns

- The mapping from placeholders to variables such that `self.instantiate(mapping) == action`, or None if no such
- *mapping exists.*

classmethod parse(*expr*)

Parse a rule expression.

Parameters
expr (str) – The string to parse, in the form `name :: [$]predicate [& [$]predicate]* -> predicate [& predicate]*`.

Return type
Rule

serialize()

Return type
Mapping

substitute(*mapping*, *name=None*)

Copy this rule, substituting certain placeholders for others.

Parameters
mapping (Mapping[*Placeholder*, *Placeholder*]) – A mapping from old to new placeholders.

Return type
Rule

property all_predicates: Iterable[*Predicate*]

All the pre- and post-conditions.

Return type
Iterable[*Predicate*]

class textworld.logic.**Signature**(*name*, *types*)

Bases: NewBase

The type signature of a Predicate or Proposition.

Create a Signature.

Parameters

- **name** (str) – The name of the proposition/predicate this signature is for.
- **types** (Iterable[str]) – The types of the parameters to the proposition/predicate.

classmethod parse(*expr*)

Parse a signature expression.

Parameters
expr (str) – The string to parse, in the form name(type [, type]*).

Return type
Signature

name

types

class textworld.logic.**State**(*logic*, *facts=None*)

Bases: object

The current state of a world.

Create a State.

Parameters

- **logic** (*GameLogic*) – The logic for this state's game.
- **facts** (*optional*) – The facts that will be true in this state.

add_fact(*prop*)

Add a fact to the state.

add_facts(*props*)

Add some facts to the state.

all_applicable_actions(*rules*, *mapping=None*)

Get all the rule instantiations that would be valid actions in this state.

Parameters

- **rules** (Iterable[*Rule*]) – The possible rules to instantiate.
- **mapping** (*optional*) – An initial mapping to start from, constraining the possible instantiations.

Return type
The actions that can be instantiated from the given rules in this state.

all_assignments(*rule*, *mapping=None*, *partial=False*, *allow_partial=None*)

Find all possible placeholder assignments that would allow a rule to be instantiated in this state.

Parameters

- **rule** (*Rule*) – The rule to instantiate.
- **mapping** (*optional*) – An initial mapping to start from, constraining the possible instantiations.
- **partial** (*optional*) – Whether incomplete mappings, that would require new variables or propositions, are allowed.
- **allow_partial** (*optional*) – A callback function that returns whether a partial match may involve the given placeholder.

Return type
Iterable[Mapping[*Placeholder*, Optional[*Variable*]]]

Returns

- *The possible mappings for instantiating the rule. Partial mappings requiring new variables will have None in*
- *place of existing Variables.*

all_instantiations(*rule*, *mapping=None*)

Find all possible actions that can be instantiated from a rule in this state.

Parameters

- **rule** (*Rule*) – The rule to instantiate.
- **mapping** (*optional*) – An initial mapping to start from, constraining the possible instantiations.

Return type
The actions that can be instantiated from the rule in this state.

apply(*action*)

Apply an action to the state.

Parameters
action (*Action*) – The action to apply.

Return type
Whether the action could be applied (i.e. whether the preconditions were met).

apply_on_copy(*action*)

Apply an action to a copy of this state.

Parameters
action (*Action*) – The action to apply.

Return type
Optional[*State*]

Returns

- The copied state after the action has been applied or None if action
- *wasn't applicable.*

are_facts(*props*)

Returns whether the propositions are all true in this state.

Return type
bool

copy()

Create a copy of this state.

Return type
State

classmethod deserialize(*data*)

Deserialize a *State* object from data.

Return type
State

facts_with_signature(*sig*)

Returns all the known facts with the given signature.

Return type
Set[*Proposition*]

has_variable(*var*)

Returns whether this state is aware of the given variable.

Return type
bool

is_applicable(*action*)

Check if an action is applicable in this state (i.e. its preconditions are met).

Return type
bool

is_fact(*prop*)

Returns whether a proposition is true in this state.

Return type
bool

is_sequence_applicable(*actions*)

Check if a sequence of actions are all applicable in this state.

Return type
bool

remove_fact(*prop*)

Remove a fact from the state.

remove_facts(*props*)

Remove some facts from the state.

serialize()

Serialize this state.

Return type
Sequence

variable_named(*name*)

Returns the variable with the given name, if known.

Return type
Variable

variables_of_type(*type*)

Returns all the known variables of the given type.

Return type
Set[*Variable*]

property facts: Iterable[*Proposition*]

All the facts in the current state.

Return type
Iterable[*Proposition*]

property variables: Iterable[*Variable*]

All the variables tracked by the current state.

Return type
Iterable[*Variable*]

class textworld.logic.**Type**(*name*, *parents*)

Bases: object

A variable type.

has_subtype_named(*name*)

Return type
bool

has_supertype_named(*name*)

Return type
bool

is_subtype_of(*other*)

Return type
bool

is_supertype_of(*other*)

Return type
bool

property ancestors: Iterable[*Type*]

The ancestors of this type (not including itself).

Return type
Iterable[*Type*]

property child_types: Iterable[*Type*]

The direct children of this type.

Return type
Iterable[*Type*]

property children: Iterable[str]

The names of the direct children of this type.

Return type
Iterable[str]

property descendants: Iterable[*Type*]

The descendants of this type (not including itself).

Return type
Iterable[*Type*]

property parent_types: Iterable[*Type*]

The parents of this type as Type objects.

Return type
Iterable[*Type*]

property subtypes: Iterable[*Type*]

This type and its descendants.

Return type
Iterable[*Type*]

property supertypes: Iterable[*Type*]

This type and its ancestors.

Return type
Iterable[*Type*]

class textworld.logic.**TypeHierarchy**

Bases: object

A hierarchy of types.

add(*type*)

closure(*type*, *expand*)

Compute the transitive closure in a type lattice according to some type relationship (generally direct sub-/super-types).

Such a lattice may look something like this:

```
  A
 / \
B   C
 \ /
  D
```

so the closure of D would be something like [B, C, A].

Return type
Iterable[*Type*]

get(*name*)

Return type
Type

multi_ancestors(*types*)

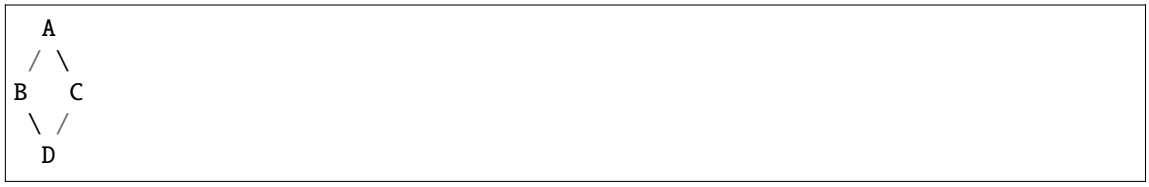
Compute the ancestral closure of a sequence of types. If these are the types of some variables, the result will be all the function parameter types that could also accept those variables.

Return type
Iterable[Collection[*Type*]]

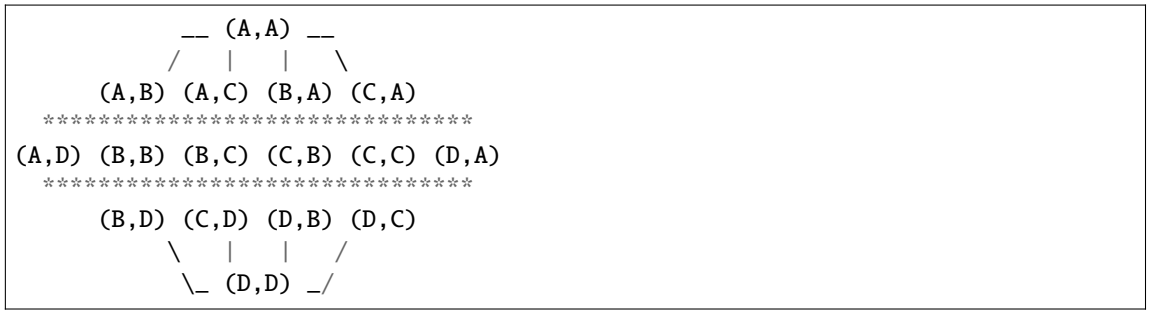
multi_closure(*types*, *expand*)

Compute the transitive closure of a sequence of types in a type lattice induced by some per-type relationship (generally direct sub-/super-types).

For a single type, such a lattice may look something like this:



so the closure of D would be something like [B, C, A]. For multiple types at once, the lattice is more complicated:



Return type
Iterable[Collection[*Type*]]

multi_descendants(*types*)

Compute the descendant closure of a sequence of types. If these are the types of some function parameters, the result will be all the variable types that could also be passed to this function.

Return type
Iterable[Collection[*Type*]]

multi_subtypes(*types*)

Computes the descendant closure of a sequence of types, including the initial types.

Return type
List[Collection[*Type*]]

multi_supertypes(*types*)

Computes the ancestral closure of a sequence of types, including the initial types.

Return type
Iterable[Collection[*Type*]]

class textworld.logic.**Variable**(*name*, *type=None*)

Bases: object

A variable representing an object in a world.

Create a Variable.

Parameters

- **name** (str) – The (unique) name of the variable.
- **type** (*optional*) – The type of the variable. Defaults to the same as the name.

classmethod deserialize(*data*)

Return type
Variable

is_a(*type*)

Return type
bool

classmethod parse(*expr*)

Parse a variable expression.

Parameters
expr (str) – The string to parse, in the form *name* or name: type.

Return type
Variable

serialize()

Return type
Mapping

name

type

class textworld.logic.model.**ActionNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

postconditions = None

preconditions = None

class textworld.logic.model.**ActionPreconditionNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

condition = None

preserve = None

class textworld.logic.model.**AliasNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

lhs = None

rhs = None

class textworld.logic.model.**ConstraintsNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

constraints = None

class textworld.logic.model.**DocumentNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

types = None

class textworld.logic.model.**GameLogicModelBuilderSemantics**(*context=None*, *types=None*)

Bases: ModelBuilderSemantics

class textworld.logic.model.**Inform7CodeNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

code = None

class textworld.logic.model.**Inform7CommandNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

command = None

event = None

rule = None

class textworld.logic.model.**Inform7CommandsNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

commands = None

class textworld.logic.model.**Inform7Node**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

parts = None

class textworld.logic.model.**Inform7PredicateNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

predicate = None

source = None

class textworld.logic.model.**Inform7PredicatesNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

predicates = None

class textworld.logic.model.**Inform7TypeNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

definition = None

kind = None

class textworld.logic.model.**ModelBase**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: Node

class textworld.logic.model.**PlaceholderNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

type = None

class textworld.logic.model.**PredicateNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

parameters = None

class textworld.logic.model.**PredicatesNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

predicates = None

class textworld.logic.model.**PropositionNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

arguments = None

name = None

class textworld.logic.model.**ReverseRuleNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

lhs = None

rhs = None

class textworld.logic.model.**ReverseRulesNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

reverse_rules = None

class textworld.logic.model.**RuleNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

postconditions = None

preconditions = None

class textworld.logic.model.**RulePreconditionNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

condition = None

preserve = None

class textworld.logic.model.**RulesNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

rules = None

class textworld.logic.model.**SignatureNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

types = None

class textworld.logic.model.**TypeNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

parts = None

supertypes = None

class textworld.logic.model.**VariableNode**(*ctx=None*, *ast=None*, *parseinfo=None*, ***kwargs*)

Bases: *ModelBase*

name = None

type = None

class textworld.logic.parser.**GameLogicBuffer**(*text*, *whitespace=None*, *nameguard=None*, *comments_re=None*, *eol_comments_re='#.*$'*, *ignorecase=None*, *namechars=''*, ***kwargs*)

Bases: Buffer

class textworld.logic.parser.**GameLogicParser**(*whitespace=None*, *nameguard=None*, *comments_re=None*, *eol_comments_re='#.*$'*, *ignorecase=None*, *left_recursion=True*, *parseinfo=True*, *keywords=None*, *namechars=''*, *buffer_class=<class 'textworld.logic.parser.GameLogicBuffer'>*, ***kwargs*)

Bases: Parser

class textworld.logic.parser.**GameLogicSemantics**

Bases: object

action(*ast*)

actionPrecondition(*ast*)

alias(*ast*)

constraints(*ast*)

document(*ast*)

inform7(*ast*)

inform7Code(*ast*)

inform7Command(*ast*)

inform7Commands(*ast*)

inform7Part(*ast*)

inform7Predicate(*ast*)

inform7Predicates(*ast*)

inform7Type(*ast*)

name(*ast*)

onlyAction(*ast*)

onlyPlaceholder(*ast*)

onlyPredicate(*ast*)

onlyProposition(*ast*)

onlyRule(*ast*)

onlySignature(*ast*)

onlyVariable(*ast*)

phName(*ast*)

placeholder(*ast*)

predName(*ast*)

predicate(*ast*)

predicateDecls(*ast*)

predicates(*ast*)

proposition(*ast*)

reverseRule(*ast*)

reverseRuleDecls(*ast*)

reverseRules(*ast*)

rule(*ast*)

ruleDecls(*ast*)

ruleName(*ast*)

rulePrecondition(*ast*)

rules(*ast*)

signature(*ast*)

signatureOrAlias(*ast*)

start(*ast*)

str(*ast*)

strBlock(*ast*)

type(*ast*)

typePart(*ast*)

variable(*ast*)

textworld.logic.parser.**main**(*filename*, *start=None*, ***kwargs*)

CHAPTER

EIGHTEEN

TEXTWORLD.RENDER

exception textworld.render.render.**WebdriverNotFoundError**

Bases: Exception

class textworld.render.render.**GraphItem**(*type*, *name*)

Bases: object

add_content(*content*)

add_unknown_predicate(*predicate*)

get_max_depth()

Returns the maximum nest depth of this plus all children. A container with no items has 1 depth, a container containing one item has 2 depth, a container containing a container which contains an item has 3 depth, and so on. :return: maximum nest depth

set_open_closed_locked(*status*)

to_dict()

property infos

class textworld.render.render.**GraphRoom**(*name*, *base_room*)

Bases: object

add_item(*item*)

Return type
None

position_string()

Return type
str

textworld.render.render.**concat_images**(**images*)

textworld.render.render.**get_webdriver**(*path=None*)

Get the driver and options objects. :param path: path to browser binary. :return: driver

textworld.render.render.**load_state**(*world*, *game_infos=None*, *action=None*, *format='png'*, *limit_player_view=False*)

Generates serialization of game state.

Parameters

- **world** (*World*) – The current state of the world to visualize.

- **game_infos** (Optional[Dict[str, *EntityInfo*]]) – The mapping needed to get objects names.
- **action** (Optional[*Action*]) – If provided, highlight the world changes made by that action.
- **format** (str) – The graph output format (gv, svg, png...)
- **limit_player_view** (bool) – Whether to limit the player's view (defaults to false)

Return type
dict

Returns
The graph generated from this World

textworld.render.render.**load_state_from_game_state**(*game_state*, *format='png'*, *limit_player_view=False*)

Generates serialization of game state.

Parameters

- **game_state** (*GameState*) – The current game state to visualize.
- **format** (str) – The graph output format (png, svg, pdf, ...)
- **limit_player_view** (bool) – Whether to limit the player's view. Default: False.

Return type
dict

Returns
The graph generated from this World

textworld.render.render.**take_screenshot**(*url*, *id='world'*)

Takes a screenshot of DOM element given its id. :type url: str :param url: URL of webpage to open headlessly. :type id: str :param id: ID of DOM element. :return: Image object.

textworld.render.render.**temp_viz**(*nodes*, *edges*, *pos*, *color=[]*)

textworld.render.render.**visualize**(*world*, *interactive=False*)

Show the current state of the world. :type world: Union[*Game*, *State*, *GameState*, *World*] :param world: Object representing a game state to be visualized. :type interactive: bool :param interactive: Whether or not to visualize the state in the browser. :return: Image object of the visualization.

textworld.render.render.**which**(*program*)

helper to see if a program is in PATH :param program: name of program :return: path of program or None

Creates server for streamed game state

class textworld.render.serve.**Server**(*game_state*, *port*)

Bases: object

Visualization server. Uses Server-sent Events to update game_state for visualization.

Note: Flask routes are defined in app.add_url_rule in order to call self in routes. :type game_state: dict :param game_state: game state returned from load_state_from_game_state :type port: int :param port: port to run visualization on

gen()

Our generator for listening for updating state. We poll for results to return us something. If nothing is returned then we just pass and keep polling. :return: yields event-stream parsed data.

index()

Index route (“/”). Returns HTML template processed by handlebars. :rtype: `str` :return: Flask response object

static listen(*conn*, *results*)

Listener for updates. Runs on separate thread. :type conn: `Connection` :param conn: child connection from multiprocessing.Pipe. :type results: `Queue` :param results: thread-safe queue for results.

start(*child_conn*)

Starts the WSGI server and listen for updates on a separate thread.

Parameters
child_conn (`Connection`) – Child connection from `multiprocessing.Pipe`.

subscribe()

Our Server-sent Event stream route. :return: A stream

update_subscribers(*game_state*)

Updates all subscribers and updates their data. This is for multiple subscribers on the visualization service. :type game_state: `dict` :param game_state: parsed game_state from load_state_from_game_state

class textworld.render.serve.**ServerSentEvent**(*data*)

Bases: `object`

Object helper to parse dict into SSE data. :type data: `any` :param data: data to pass to SSE

encode()

class textworld.render.serve.**SupressStdStreams**

Bases: `object`

for surpressing std.out streams

class textworld.render.serve.**VisualizationService**(*game_state*, *open_automatically*)

Bases: `object`

Server for visualization.

We instantiate a new process for our flask server, so our game can send updates to the server. The server instantiates new gevent Queues for every connection.

start(*parent_thread*, *port*)

Start visualization server on a new process. :type parent_thread: `Thread` :param parent_thread: the parent thread that called start. :type port: `int` :param port: Port to run visualization on.

Return type
`None`

start_server(*game_state*, *port*, *child_conn*)

function for starting new server on new process. :type game_state: `dict` :param game_state: initial game state from load :type port: `int` :param port: port to run server :type child_conn: `Connection` :param child_conn: child connection from multiprocessing.Pipe

stop_server()

update_state(*game_state*, *command*)

Propogate state update to server. We use a multiprocessing.Pipe to pass state into flask process. :type game_state: *GameState* :param game_state: Glulx game state. :type command: `str` :param command: previous command

textworld.render.serve.**find_free_port**(*port_range*)

textworld.render.serve.**get_html_template**(*game_state=None*)

CHAPTER

NINETEEN

TEXTWORLD.UTILS

class textworld.utils.**RandomGenerator**(*seed=None*)

Bases: object

Random generator controlling the games generation.

next()

Start a new random generator using a new seed.

set_seed(*seed*)

property seed

class textworld.utils.**RegexDict**

Bases: OrderedDict

Ordered dictionary that supports querying with regex.

References

Adapted from https://stackoverflow.com/questions/21024822/python-accessing-dictionary-with-wildcards.

get_matching(**regexes*, *exclude=[]*)

Query the dictionary using one or several regular expressions.

Parameters

- ***regexes** – List of regular expressions determining which keys of this dictionary are relevant to this query.
- **exclude** (List[str]) – List of regular expressions determining which keys of this dictionary should be excluded from this query.

Return type
List[Any]

Returns
The value associated to each relevant (and not excluded) keys.

textworld.utils.**check_modules**(*required_modules*, *missing_modules*)

Check whether some required modules are missing.

textworld.utils.**chunk**(*iterable*, *n*, *fct=<function <lambda>>*)

Return type
Iterable[Iterable]

textworld.utils.**encode_seeds**(*seeds*)

Generate UID from a list of seeds.

textworld.utils.**make_temp_directory**(*suffix=''*, *prefix='tw_'*, *dir=None*)

Create temporary folder to used in a with statement.

textworld.utils.**maybe_mkdir**(*dirpath*)

Create all parent folders if needed.

textworld.utils.**save_graph_to_svg**(*G*, *labels*, *filename*, *backward=False*)

Generate a figure of a networkx's graph object and save it.

textworld.utils.**str2bool**(*v*)

Convert string to a boolean value. .. rubric:: References

https://stackoverflow.com/questions/715417/converting-from-a-string-to-boolean-in-python/715468#715468

textworld.utils.**take**(*n*, *iterable*)

Return first n items of the iterable as a list.

References

https://docs.python.org/3/library/itertools.html#itertools-recipes

Return type
Iterable

textworld.utils.**unique_product**(**iterables*)

Cartesian product of input iterables with pruning.

This method prunes any product tuple with duplicate elements in it.

Example

unique_product('ABC', 'Ax', 'xy') –> Axy BAx BAy Bxy CAx CAy Cxy

Notes

This method is faster than the following equivalent code:

```
>>> for result in itertools.product(*args):
>>>     if len(set(result)) == len(result):
>>>         yield result
```

textworld.utils.**uniquify**(*seq*)

Order preserving uniquify.

References

Made by Dave Kirby https://www.peterbe.com/plog/uniqifiers-benchmark

textworld.utils.**g_rng = <textworld.utils.RandomGenerator object>**

Global random generator.

CHAPTER

TWENTY

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Parisotto2017] Emilio Parisotto and Ruslan Salakhutdinov. Neural map: Structured memory for deep reinforcement learning. arXiv:1702.08360, 2017.

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X