TextWorld Documentation

Release 1.1.0-5-g546d44d

Marc-Alexandre Côté, Tavian Barnes, Matthew Hausknecht, James

Feb 08, 2019

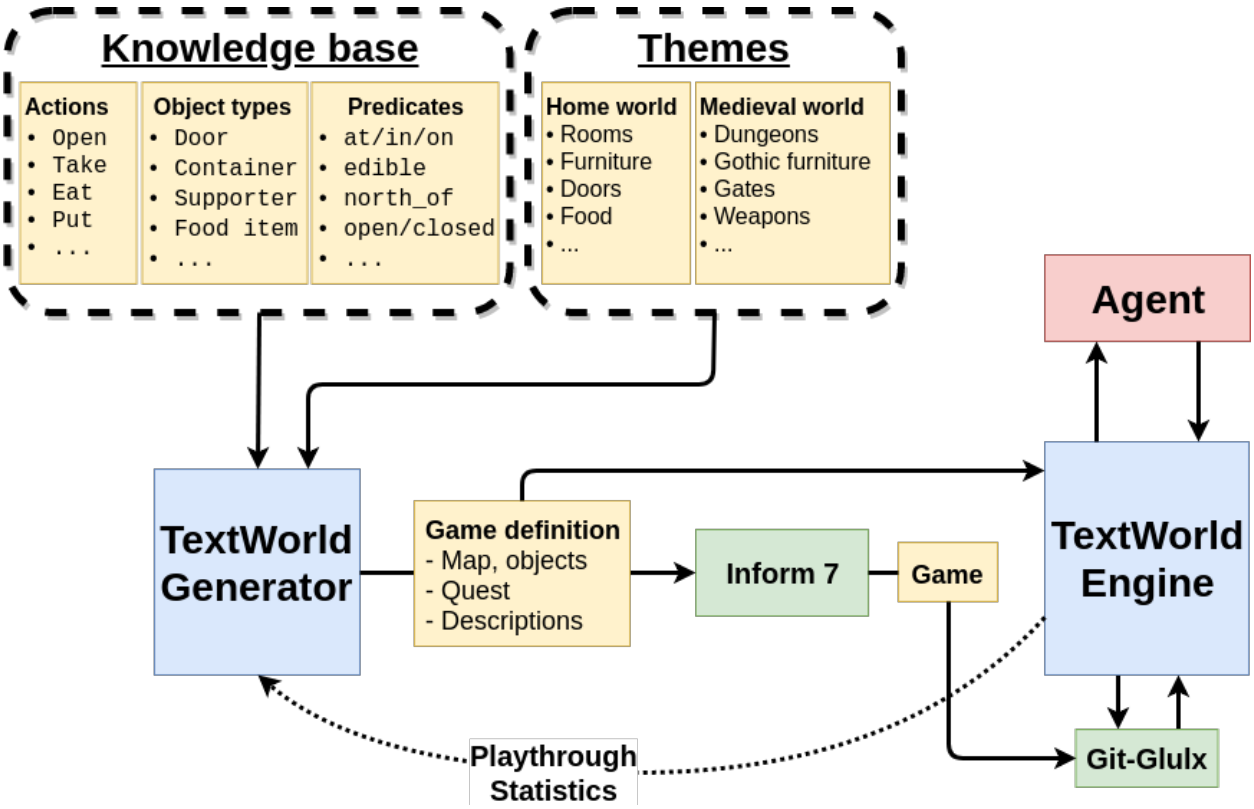
Notes

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

CHAPTER 1

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 2

Custom Environment

Making new Environments is useful when games output more information than needed (e.g. a header). It is also useful for detecting when a game ends (either win or lose) since most games have different ways of letting the player knows the game has ended.

This page will guide you through creating a new environment. You should have prior knowledge of the *textworld.core.Environment* and *textworld.core.GameState* classes.

2.1 Zork1Environment

For this tutorial, we will be creating an Environment dedicated to Zork1. You can get a copy of the game from the Internet Archive).

2.1.1 Why do we want a new environment?

Let's start by running the game directly using the native frotz interpreter as follows

```
dfrotz path/to/zork1.z5
```

As we can see (see left column in the table below), Zork1 prints out a header displaying the name of the room, the scores and the number of moves. Also, when we start the game, some version numbers and other information are printed.

All this decorative text is just noise really. One could argue that header contains meaningful information but the `game_state` object the agent receives already should have all that information. So, there is not point in making the task more difficult for the agent than it already is.

Frotz	TextWorld
West of House Score: 0 Moves: 0 ZORK I: The Great Underground Empire Copyright (c) 1981, 1982, 1983 Infocom, Inc. All rights reserved. ZORK is a registered trademark of Infocom, Inc. Revision 88 / Serial number 840726 West of House You are standing in an open field west of a white house, with a boarded front door. There is a small mailbox here. >open mailbox West of House Score: 0 Moves: 1 Opening the small mailbox reveals a leaflet.	West of House You are standing in an open field west of a white house, with a boarded front door. There is a small mailbox here. > open mailbox Opening the small mailbox reveals a leaflet.

As we can see in the right column of the above table, the input text is more compact.

2.1.2 Building the Zork1Environment

Since `zork1.z5` is a Z-Machine game, we want to leverage the *python-frotz* communication pipeline already in place in *`textworld.envs.FrotzEnvironment`*. To do so, we simply have to subclass it.

```
from textworld.envs import FrotzEnvironment

class Zork1Environment(FrotzEnvironment):
    GAME_STATE_CLASS = Zork1GameState
```

As we can see there is nothing much to modify for this particular environment other than specifying the game state class to use. This is because we only need to clean the output text rather than changing some fundamental behavior of the `FrotzEnvironment`.

Cleaning up the output is essentially done in the *`Zork1GameState`* (a subclass of *`GameState`*) using a bunch of regular expressions. Here's what it looks like

```
import textworld

class Zork1GameState(textworld.GameState):

    def _remove_header(self, text):
        cleaned_text = text_utils.remove_header(text)
        return cleaned_text.lstrip("\n")

    def _check_for_death(self, text):
        return "****  You have died  ****" in text

    @property
    def nb_deaths(self):
        """ Number of times the player has died. """
        if not hasattr(self, "_nb_deaths"):
            if self.previous_state is None:
                self._nb_deaths = 0
            else:
                has_died = self._check_for_death(self.feedback)
                self._nb_deaths = self.previous_state.nb_deaths + has_died

        return self._nb_deaths
```

(continues on next page)

(continued from previous page)

```
    @property
    def feedback(self):
        """ Interpreter's response after issuing last command. """
        if not hasattr(self, "_feedback"):
            # Extract feeback from command's output.
            self._feedback = self._remove_header(self._raw)
            if self.previous_state is None:
                # Remove version number and copyright text.
                self._feedback = "\n".join(self._feedback.split("\n")[5:])

        return self._feedback

    @property
    def inventory(self):
        """ Player's inventory. """
        if not hasattr(self, "_inventory"):
            # Issue the "inventory" command and parse its output.
            text = self._env.send("inventory")
            self._inventory = self._remove_header(text)

        return self._inventory

    def _retrieve_score(self):
        if self.has_won or self.has_lost:
            _score_text = self.feedback
        else:
            # Issue the "score" command and parse its output.
            text = self._env.send("score")
            _score_text = self._remove_header(text)

        regex = r"Your score is (?P<score>[0-9]+) \(total of (?P<max_score>[0-9]+)␣
↪points\)"
        match = re.match(regex, _score_text)
        self._score = int(match.groupdict()['score'].strip())
        self._max_score = int(match.groupdict()['max_score'].strip())
        return self._score, self._max_score

    @property
    def score(self):
        """ Current score. """
        if not hasattr(self, "_score"):
            self._retrieve_score()

        return self._score

    @property
    def max_score(self):
        """ Max score for this game. """
        if not hasattr(self, "_max_score"):
            self._retrieve_score()

        return self._max_score

    @property
    def description(self):
        """ Description of the current location. """
```

(continues on next page)

(continued from previous page)

```
        if not hasattr(self, "_description"):
            # Issue the "look" command and parse its output.
            text = self._env.send("look")
            self._description = self._remove_header(text)

        return self._description

    @property
    def has_won(self):
        """ Whether the player has won the game or not. """
        return "Inside the Barrow" in self.feedback.split("\n")[0]

    @property
    def has_lost(self):
        """ Whether the player has lost the game or not. """
        return self.nb_deaths >= 3
```

Then the last thing to do is to make TextWorld framework aware of that new environment. This is done by adding a new entry to the `CUSTOM_ENVIRONMENTS` dictionary located in `textworld/envs/__init__.py`.

```
# Import dedicated environment
from textworld.envs.frotz.zork1 import Zork1Environment

CUSTOM_ENVIRONMENTS = {
    "zork1.z5": Zork1Environment
}
```

With everything in place, we can check the results using `tw-play zork.z5`.

CHAPTER 3

Known Issues

3.1 Inform 7

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

3.2 FrotzEnvironment

This is known to cause some concurrency issues when commands are rapidely sent to the game's interpreter. An alternative is to use the *JerichoEnvironment* for Z-Machine games or `JerichoEnvironment` for games generated with TextWorld.

CHAPTER 4

textworld

4.1 Core

exception textworld.core.**GameNotRunningError**
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

class textworld.core.**Environment**
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.
```

activate_state_tracking()
Enables state tracking.

Return type None

close()
Ends the game.

Return type None

compute_intermediate_reward()
Enables intermediate reward computation.

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.

display_command_during_render
Enables/disables displaying the command when rendering.

Return type bool

metadata
Environment's metadata.

For instance, it can contain the supported rendering modes `'render.modes': {'human', 'text', 'ansi'}`.

Return type Mapping[~KT, +VT_co]

class textworld.core.**GameState**(*env=None*)
Bases: object

Representation of the state of a text-based game.

This object can be used to get additional information about the current state of the game.

Create a game state.

Parameters **env** (Optional[*Environment*]) – Environment that can be used to fetch additional information.

init(*output*)
Initializes the game state from intro text.

Parameters **output** (str) – Text displayed when the game starts.

Return type None

update (*command*, *output*)
Creates a new game state with the new information.

Parameters

- **command** (str) – Command sent to the game's interpreter.
- **output** (str) – Response from the game's interpreter.

Returns The new state of the game.

Return type *GameState*

command
Last command sent to the interpreter.

Return type str

description
Description at the current location.

It's usually the output of the "look" command.

Return type str

feedback
Interpreter's response after issuing last command.

Return type str

game_ended
Whether the game is finished or not.

Return type bool

has_lost
Whether the player has lost the game or not.

Return type bool

has_won
Whether the player has won the game or not.

Return type bool

inventory
Player's inventory.

It's usually the output of the "inventory" command.

Return type str

location
Name of the current location.

Return type str

max_score
Max score for this game.

It's usually the output of the "score" command.

Return type float

nb_moves
Number of actions perfomed up until now.

Return type int

score
Current score.

It's usually the output of the "score" command.

Return type float

class textworld.core.**Wrapper**(*env*)
Bases: *textworld.core.Environment*

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** (*Environment*) – environment to wrap.

activate_state_tracking()
Enables state tracking.

Return type None

close()
Ends the game.

Return type None

compute_intermediate_reward()
Enables intermediate reward computation.

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[Any]

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 List[int]

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.

display_command_during_render
Enables/disables displaying the command when rendering.

Return type bool

metadata

Environment's metadata.

For instance, it can contain the supported rendering modes `'render.modes': {'human', 'text', 'ansi'}`.

Return type `Mapping`[~KT, +VT_co]

CHAPTER 5

textworld.gym

textworld.gym.utils.**make_batch**(*env_id*, *batch_size*, *parallel=False*)

Make an environment that runs multiple games independently.

Parameters

- **env_id** (str) – Environment ID that will compose a batch.
- **batch_size** (int) – Number of independent environments to run.
- **parallel** (bool) – If True, the environment will be executed in different processes.

Return type str

Returns The corresponding gym-compatible env_id to use.

textworld.gym.utils.**register_game**(*game_file*, *request_infos=None*, *max_episode_steps=50*, *name=''*)

Make an environment for a particular game.

Parameters

- **game_file** (str) – Path for the TextWorld game (.ulx).
- **request_infos** (Optional[*EnvInfos*]) – For customizing the information returned by this environment (see *textworld.EnvInfos* for the list of available information).
- **max_episode_steps** (int) – Terminate a game after that many steps.
- **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."
>>> game_file = 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(game_file, request_infos)
>>> env = gym.make(env_id)
>>> ob, infos = env.reset()
>>> print(infos["extra.more"])
This is extra information.
```

textworld.gym.utils.**register_games**(*game_files*, *request_infos=None*, *max_episode_steps=50*, *name=''*)

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

Parameters

- **game_files** (List[str]) – Paths for the TextWorld games (.ulx).
- **request_infos** (Optional[*EnvInfos*]) – For customizing the information returned by this environment (see *textworld.EnvInfos* for the list of available information).
- **max_episode_steps** (int) – Terminate a game after that many steps.
- **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."
>>> game_file = 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([game_file], request_infos)
>>> env = gym.make(env_id)
>>> ob, infos = env.reset()
>>> print(infos["extra.more"])
This is extra information.
```

5.1 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.

infos_to_request
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*

5.2 Envs

class textworld.gym.envs.textworld_games_env.**TextworldGamesEnv**(*game_files*, *request_infos=None*, *action_space=None*, *observation_space=None*)

Bases: gym.core.Env

Environment for playing TextWorld games.

Each time *TextworldGamesEnv.reset()* is called, a new game from the pool starts. Each game of the pool is guaranteed to be played exactly once before a same game is played for a second time.

Parameters

- **game_files** (List[str]) – Paths of every TextWorld game composing the pool (.ulx + .json).
- **request_infos** (Optional[*EnvInfos*]) – For customizing the information returned by this environment (see *textworld.EnvInfos* for the list of available information).
- **action_space** (Optional[Space]) – The action space of this TextWorld environment. By default, a *textworld.gym.spaces.Word* instance is used with a max_length of 8 and a vocabulary extracted from the TextWorld game.
- **observation_space** (Optional[Space]) – The observation space of this TextWorld environment. By default, a *textworld.gym.spaces.Word* instance is used with a max_length of 200 and a vocabulary extracted from the TextWorld game.

close()

Close this environment.

Return type None

render(*mode='human'*)

Renders the current state of this environment.

The rendering is composed of the previous text command (if there's one) and the text describing the current observation.

Parameters **mode** (str) – Controls where and how the text is rendered. Supported modes are:

- human: Display text to the current display or terminal and return nothing.
- ansi: Return a StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).
- text: Return a string (str) containing the text without any ANSI escape sequences.

Return type Union[StringIO, str, None]

Returns Depending on the mode, this method returns either nothing, a string, or a StringIO object.

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.

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(*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.

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

class textworld.gym.envs.batch_env.**BatchEnv**(*env_id*, *batch_size*)

Bases: gym.core.Env

Environment to run multiple games independently.

Parameters

- **env_id**(*list of str or str*) – Environment IDs that will compose a batch. If only one env_id is provided, it will be repeated batch_size times.
- **batch_size** (*int*) – Number of independent environments to run.

close()

Override _close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

render(*mode='human'*)

Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

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.

Parameters

- **mode** (*str*) – the mode to render with
- **close** (*bool*) – close all open renderings

Example:

class MyEnv(Env): metadata = {'render.modes': ['human', 'rgb_array']}

def render(self, mode='human'):

if mode == 'rgb_array': return np.array(...) # return RGB frame suitable for video

elif mode is 'human': ... # pop up a window and render

else: super(MyEnv, self).render(mode=mode) # just raise an exception

reset()

Reset all environments of the batch.

Returns Text observations, i.e. command's feedback. infos: Information requested when creating the environments.

Return type obs

seed(*seed=None*)

Sets the seed for this env's random number generator(s).

Note: Some environments use multiple pseudorandom number generators. We want to capture all such seeds used in order to ensure that there aren't accidental correlations between multiple generators.

Returns

Returns the list of seeds used in this env's random number generators. The first value in the list should be the "main" seed, or the value which a reproducer should pass to 'seed'. Often, the main seed equals the provided 'seed', but this won't be true if seed=None, for example.

Return type list<bigint>

skip(*ngames=1*)

step(*actions*)
Perform one action per environment of the batch.

Returns Text observations, i.e. command's feedback. reward: Current game score. done: Whether the game is over or not. infos: Information requested when creating the environments.

Return type obs

class textworld.gym.envs.batch_env.**ParallelBatchEnv**(*env_id*, *batch_size*)
Bases: gym.core.Env

Environment to run multiple games in parallel.

Parameters

- **env_id**(*list of str or str*) – Environment IDs that will compose a batch. If only one env_id is provided, it will be repeated batch_size times.
- **batch_size** (*int*) – Number of environment to run in parallel.

close()
Override _close in your subclass to perform any necessary cleanup.

Environments will automatically close() themselves when garbage collected or when the program exits.

render(*mode='human'*)
Renders the environment.

The set of supported modes varies per environment. (And some environments do not support rendering at all.) By convention, if mode is:

- human: render to the current display or terminal and return nothing. Usually for human consumption.
- rgb_array: Return an numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video.
- ansi: Return a string (str) or StringIO.StringIO containing a terminal-style text representation. The text can include newlines and ANSI escape sequences (e.g. for colors).

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.

Parameters

- **mode** (*str*) – the mode to render with
- **close** (*bool*) – close all open renderings

Example:

class MyEnv(Env): metadata = {'render.modes': ['human', 'rgb_array']}

def render(self, mode='human'):

if mode == 'rgb_array': return np.array(...) # return RGB frame suitable for video

elif mode is 'human': ... # pop up a window and render

else: super(MyEnv, self).render(mode=mode) # just raise an exception

reset()
Reset all environments of the batch.

Returns Text observations, i.e. command's feedback. infos: Information requested when creating the environments.

Return type obs

seed(*seed=None*)
Sets the seed for this env's random number generator(s).

Note: Some environments use multiple pseudorandom number generators. We want to capture all such seeds used in order to ensure that there aren't accidental correlations between multiple generators.

Returns

Returns the list of seeds used in this env's random number generators. The first value in the list should be the "main" seed, or the value which a reproducer should pass to 'seed'. Often, the main seed equals the provided 'seed', but this won't be true if seed=None, for example.

Return type list<bigint>

skip(*ngames=1*)

step(*actions*)
Perform one action per environment of the batch.

Returns Text observations, i.e. command's feedback. reward: Current game score. done: Whether the game is over or not. infos: Information requested when creating the environments.

Return type obs

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]`

5.3 Spaces

exception textworld.gym.spaces.text_spaces.**VocabularyHasDuplicateTokens**
Bases: `ValueError`

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

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 6

textworld.envs

6.1 Glulx

exception textworld.envs.glulx.git_glulx_ml.**ExtraInfosIsMissingError**(*info*)
Bases: NameError

Thrown if extra information is required without enabling it first via `tw-extra-infos CMD`.

exception textworld.envs.glulx.git_glulx_ml.**MissingGameInfosError**
Bases: NameError

Thrown if an action requiring GameInfos is used on a game without GameInfos, such as a Frotz game or a Glulx game not generated by TextWorld.

exception textworld.envs.glulx.git_glulx_ml.**OraclePolicyIsRequiredError**(*info*)
Bases: NameError

Thrown if an action requiring an Oracle-based reward policy is called without the intermediate reward being active.

exception textworld.envs.glulx.git_glulx_ml.**StateTrackingIsRequiredError**(*info*)
Bases: NameError

Thrown if an action requiring state tracking is performed while state tracking is not enabled.

class textworld.envs.glulx.git_glulx_ml.**GitGlulxMLEnvironment**(*gamefile*)
Bases: *textworld.core.Environment*

Environment to support playing Glulx games generated by TextWorld.

TextWorld 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 message queues.

Creates a GitGlulxML from the given gamefile

Parameters **gamefile** (str) – The name of the gamefile to load.

activate_state_tracking()
Enables state tracking.

Return type None

close()
Ends the game.

Return type None

compute_intermediate_reward()
Enables intermediate reward computation.

Return type None

enable_extra_info(*info*)

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 *GlulxGameState*

Returns Initial state of the game.

step(*command*)
Performs a given command.

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

Return type Tuple[*GlulxGameState*, 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.

game_running
Determines if the game is still running.

Return type bool

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

class textworld.envs.glulx.git_glulx_ml.**GlulxGameState**(**args*, ***kwargs*)
Bases: *textworld.core.GameState*

Encapsulates the state of a Glulx game. This is the primary interface to the Glulx game driver.

Takes the same parameters as textworld.GameState :param args: The arguments :param kwargs: The kwargs

init(*output*, *game*, *state_tracking=False*, *compute_intermediate_reward=False*)
Initialize the game state and set tracking parameters. The tracking parameters, state_tracking and compute_intermediate_reward, are computationally expensive, so are disabled by default.

Parameters

- **output** (str) – Introduction text displayed when a game starts.
- **game** (*Game*) – The glulx game to run

- **state_tracking** (bool) – Whether to use state tracking
- **compute_intermediate_reward** (bool) – Whether to compute the intermediate reward

update(*command*, *output*)
Updates the GameState with the command from the agent and the output from the interpreter. :type command: str :param command: The command sent to the interpreter :type output: str :param output: The output from the interpreter :rtype: *GlulxGameState* :return: A GameState of the current state

view()
Returns a view of this Game as a GameState :rtype: *GlulxGameState* :return: A GameState reflecting the current state

action
Last action that was detected.

Return type *Action*

admissible_commands
Return the list of admissible commands given the current state.

command_feedback
Return the parser response related to the previous command.

This corresponds to the feedback without the room description, the inventory and the objective (if they are present).

command_templates

description
Description at the current location.

It's usually the output of the "look" command.

entities

extras

game_ended
Whether the game is finished or not.

Return type bool

game_infos
Additional information about the game.

Return type Mapping[~KT, +VT_co]

has_lost
Whether the player has lost the game or not.

has_won
Whether the player has won the game or not.

intermediate_reward
Reward indicating how useful the last action was for solving the quest.

inventory
Player's inventory.

It's usually the output of the "inventory" command.

max_score
Max score for this game.

It's usually the output of the "score" command.

objective
Objective of the game.

policy_commands
Commands to entered in order to complete the quest.

score
Current score.

It's usually the output of the "score" command.

state
Current game state.

Return type *State*

verbs

6.2 Wrappers

class textworld.envs.wrappers.recorder.**Recorder**
Bases: *textworld.core.Wrapper*

reset()
Starts game from the beginning.

Return type *GameState*

Returns Initial state of the game.

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.

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

Returns

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.

class textworld.envs.wrappers.filter.**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 information specific to certain games.

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

basics
Information requested excluding the extras.

Return type Iterable[str]

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

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

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

extras
List[str] – Names of extra information which are game specific.

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

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

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

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

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

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

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

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

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

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

class textworld.envs.wrappers.filter.**Filter**(*options*)
Bases: *textworld.core.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 = Filter(env)
>>> ob, infos = env.reset()
>>> print(infos["description"])
>>> print(infos["inventory"])
>>> print(infos["extra.more"])
```

Parameters **options** (*EnvInfos*) – For customizing the information returned by this environment (see *textworld.EnvInfos* for the list of available information).

reset()
Starts game from the beginning.

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

Returns Initial state of the game.

step(*command*)
Performs a given command.

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

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

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.

6.3 Z-Machine

class textworld.envs.zmachine.frotz.**DefaultZGameState**(*env=None*)
Bases: *textworld.core.GameState*

Create a game state.

Parameters **env** (Optional[*Environment*]) – Environment that can be used to fetch additional information.

description
Description of the current location.

feedback
Interpreter's response after issuing last command.

has_lost
Whether the player has lost the game or not.

has_won
Whether the player has won the game or not.

inventory
Player's inventory.

max_score
Max score for this game.

nb_deaths
Number of times the player has died.

score
Current score.

class textworld.envs.zmachine.frotz.**FrotzEnvironment**(*game_filename*)
Bases: *textworld.core.Environment*

Environment to support playing Z-Machine games.

FrotzEnvironment relies on the frotz interpreter which is started in a seperate process (using the Python subprocess module. Then, the *FrotzEnvironment* sends text commands via stdin and reads the output from stdout. This is known to cause some concurrency issues. For that reason textworld.envs.JerichoEnviroment is preferred.

Parameters **game_filename** (str) – Path to the game file.

GAME_STATE_CLASS
alias of *DefaultZGameState*

close()
Ends the game.

Return type None

render(*mode='human'*)
Renders the current state of the game.

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

reset()
Starts game from the beginning.

Return type *DefaultZGameState*

Returns Initial state of the game.

seed(*seed*)
Sets the seed for the random number generator.

Return type List[~T]

send(*command*)
Send a command to the game and return the output.

Parameters **command** (*str*) – Command to senf to Frotz (Z-machine's interpreter).

Returns The feeback message of the command sent.

Return type str

step(*command*)
Performs a given command.

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

Return type Tuple[*DefaultZGameState*, 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.

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

exception textworld.envs.zmachine.jericho.**JerichoUnsupportedGameWarning**
Bases: UserWarning

class textworld.envs.zmachine.jericho.**JerichoEnvironment**(*game_filename*)
Bases: *textworld.core.Environment*

Parameters **game_filename** (*str*) – The game's filename.

GAME_STATE_CLASS
alias of *JerichoGameState*

close()
Ends the game.

render(*mode='human'*, *close=False*)
Renders the current state of the game.

Parameters **mode** – The mode to use for rendering.

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.

game_running
Determines if the game is still running.

Return type bool

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

class textworld.envs.zmachine.jericho.**JerichoGameState**(*env=None*)
Bases: *textworld.core.GameState*

Create a game state.

Parameters **env** (Optional[*Environment*]) – Environment that can be used to fetch additional information.

description
Description of the current location.

feedback
Interpreter's response after issuing last command.

has_lost
Whether the player has lost the game or not.

has_won
Whether the player has won the game or not.

inventory
Player's inventory.

max_score
Max score for this game.

nb_deaths
Number of times the player has died.

score
Current score.

class textworld.envs.zmachine.zork1.**Zork1Environment**(*game_filename*)
Bases: *textworld.envs.zmachine.frotz.FrotzEnvironment*

Parameters **game_filename** (str) – Path to the game file.

GAME_STATE_CLASS
alias of *Zork1GameState*

class textworld.envs.zmachine.zork1.**Zork1GameState**(*env=None*)
Bases: *textworld.core.GameState*

Create a game state.

Parameters **env** (Optional[*Environment*]) – Environment that can be used to fetch additional information.

description
Description of the current location.

feedback
Interpreter's response after issuing last command.

has_lost
Whether the player has lost the game or not.

has_won
Whether the player has won the game or not.

inventory
Player's inventory.

max_score
Max score for this game.

nb_deaths
Number of times the player has died.

score
Current score.

CHAPTER 7

textworld.agents

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

Bases: *textworld.core.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: *textworld.core.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: *textworld.core.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: *textworld.core.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: *textworld.core.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 8

textworld.generator

exception textworld.generator.**GenerationWarning**
Bases: UserWarning

exception textworld.generator.**NoSuchQuestExistError**
Bases: NameError

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'.

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.

fixed_mapping
A fixed mapping from placeholders to variables, for singletons.

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

logic
The rules of the game.

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.

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.

fixed_mapping

Return type *GameLogic*

logic

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.

class textworld.generator.dependency_tree.**DependencyTree**(*element_type=<class 'textworld.generator.dependency_tree.Dependency*
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.

empty

Return type bool

leaves_elements

Return type List[*DependencyTreeElement*]

leaves_values

Return type List[Any]

values

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()

User interface using urwid

class textworld.generator.user_query.**UrwidQuestQuerier**(*checkboxes*)
Bases: urwid.wimp.PopUpLauncher

UI class for query_for_important_quests

cancel_button_clicked(_)

confirm_button_clicked(_)

create_pop_up()
Subclass must override this method and return a widget to be used for the pop-up. This method is called once each time the pop-up is opened.

get_pop_up_parameters()
Subclass must override this method and have it return a dict, eg:

{'left':0, 'top':1, 'overlay_width':30, 'overlay_height':4}

This method is called each time this widget is rendered.

class textworld.generator.user_query.**UrwidWarningDialog**(*msg_text*)
Bases: urwid.widget.WidgetWrap

Generic message dialog with no text, suitable for warnings.

signals = ['close']

textworld.generator.user_query.**query_for_important_facts**(*actions*, *facts=None*, *var_infos=None*)
Queries the user, asking which facts are important.

Parameters

- **actions** (List[*Action*]) – Actions used to determine or extract relevant facts.
- **facts** (Optional[List[*Proposition*]]) – All facts existing at the end of the game.

Return type Optional[List[*Proposition*]]

Returns The list of facts that are required to win; or None if facts was not provided; or None if the user cancels.

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[~T]

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.

8.1 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: *textworld.generator.dependency_tree.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 Optional[*Action*]

Returns Whether the tree has changed or not.

class textworld.generator.game.**ActionDependencyTreeElement**(*value*)
Bases: *textworld.generator.dependency_tree.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 preconditions 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

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[~KT, +VT_co]) – Serialized data with the needed information to build a *EntityInfo* object.

Return type *EntityInfo*

serialize()
Serialize this object.

Results: EntityInfo's data serialized to be JSON compatible

Return type Mapping[~KT, +VT_co]

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

definite
str – The definite article to use for this entity.

desc
str – Text description displayed when examining this entity in the game.

id
str – Unique name for this entity. It is used when generating

indefinite
str – The indefinite article to use for this entity.

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

noun
str – The noun part of the name, if available.

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

synonyms
List[str] – Alternative names that can be used to refer to this entity.

type
str – The type of this entity.

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.

actions
Actions to be performed to trigger this event

commands
Human readable version of the actions.

condition
textworld.logic.Action that can only be applied when all conditions are 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[~KT, +VT_co]) – 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[~KT, +VT_co]

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.

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.

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

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

Return type bool

triggered
Check whether the event has been triggered.

Return type bool

triggering_policy
Actions to be performed in order to trigger the event.

Return type List[*Action*]

untriggerable
Check whether the event is in an untriggerable state.

Return type bool

class textworld.generator.game.**Game**(*world*, *grammar=None*, *quests=()*, *kb=None*)
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[~KT, +VT_co]) – 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[~KT, +VT_co]

command_templates
All command templates understood in this game.

Return type List[str]

directions_names

Return type List[str]

entity_names

Return type List[str]

infos
Information about the entities in the game.

Return type Dict[str, *EntityInfo*]

objective

Return type str

objects_names
The names of all relevant objects in this game.

Return type List[str]

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

Return type List[str]

objects_types
All types of objects in this game.

Return type List[str]

verbs
Verbs that should be recognized in this game.

Return type List[str]

win_condition
All win conditions, one for each quest.

Return type List[Collection[*Proposition*]]

class textworld.generator.game.**GameOptions**
Bases: object

Options for customizing the game generation.

nb_rooms
int – Number of rooms in the game.

nb_objects
int – Number of objects in the game.

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

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

quest_breadth
int – Number of subquests per independent quest. It controls how nonlinear a quest can be (1: linear).

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

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

force_recompile
bool – If `True`, recompile game even if it already exists.

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

seeds
Optional[Union[int, Dict]] –

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*).

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

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

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

copy()

Return type *GameOptions*

kb

Return type *KnowledgeBase*

quest_breadth

Return type int

quest_length

Return type int

rngs

Return type Dict[str, RandomState]

seeds

uuid

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.

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

completed
Whether all quests are completed.

Return type bool

done
Whether all quests are completed or at least one has failed or is unfinishable.

Return type bool

failed
Whether at least one quest has failed or is unfinishable.

Return type bool

max_score
Sum of the reward of all quests.

Return type int

score
Sum of the reward of all completed quests.

Return type int

tracking_quests
Whether quests are being tracked or not.

Return type bool

valid_actions
Actions that are valid at the current state.

Return type List[*Action*]

winning_policy
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=()*)

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.

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

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

reward
Reward given for completing this quest.

desc
A text description of the quest.

commands
List of text commands leading to this quest completion.

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.

copy()
Copy this quest.

Return type *Quest*

classmethod deserialize(*data*)
Creates a *Quest* from serialized data.

Parameters **data** (Mapping[~KT, +VT_co]) – 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.

Results: Quest's data serialized to be JSON compatible

Return type Mapping[~KT, +VT_co]

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.

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

completed
Check whether the quest is completed.

Return type bool

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

Return type bool

failed
Check whether the quest has failed.

Return type bool

unfinishable
Check whether the quest is in an unfinishable state.

Return type bool

winning_policy
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]

8.2 World

exception textworld.generator.world.**NoFreeExitError**
Bases: Exception

class textworld.generator.world.**World**
Bases: object

add_fact(*fact*)

Return type None

add_facts(*facts*)

Return type None

classmethod deserialize(*serialized_facts*)

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*)

Return type *World*

classmethod from_map(*map*)

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[~T]

set_player_room(*start_room=None*)

Return type None

entities

Return type Valuesview[*WorldEntity*]

facts

Return type List[*Proposition*]

objects

Return type List[*WorldObject*]

player_room

Return type *WorldRoom*

rooms

Return type List[*WorldRoom*]

state

Return type *State*

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

Bases: *textworld.logic.Variable*

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

add_related_fact(*fact*)

Return type None

classmethod create(*var*)

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

get_attributes()

Return type List[*Proposition*]

id

Return type str

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

Bases: *textworld.generator.world.WorldEntity*

A WorldObject is anything we can directly interact with.

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

Bases: *textworld.generator.world.WorldEntity*

WorldRooms can be linked with each other through exits.

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 cartesian graph on a grid.

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*)

8.3 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

class textworld.generator.maker.**GameMaker**
Bases: object

Stateful utility class for handcrafting text-based games.

player
WorldEntity – Entity representing the player.

inventory
WorldEntity – Player's envi entity.

rooms
List[WorldRoom] – The rooms present in this world.

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

add_distractors(*nb_distractors*)
Adds a number of distractors - random objects.

Parameters **nb_distractors** (int) – The number of distractors to add.

Return type None

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

add_random_quest(*max_length*)
Generates a random quest for the game.

Calling this method replaced all previous quests.

Parameters **max_length** (int) – The maximum length of the quest to generate.

Return type *Quest*

Returns The generated quest.

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.

Returns

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.

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.

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(*ask_for_state=False*)
Defines the game's quest by recording the commands.

This launches a `textworld.play` session.

Parameters **ask_for_state** (`bool`) – If true, the user will be asked to specify which set of facts of the final state are should be true in order to consider the quest as completed.

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*, *ask_for_state=False*)
Defines the game's quest using predefined text commands.

This launches a `textworld.play` session.

Parameters

- **commands** (List[str]) – Text commands.
- **ask_for_state** (bool) – If true, the user will be asked to specify which set of facts of the final state are should be true in order to consider the quest as completed.

Return type *Quest*

Returns The resulting quest.

test()
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

facts
All the facts associated to the current game state.

Return type Iterable[*Proposition*]

state
Current state of the world.

Return type *State*

class textworld.generator.maker.**WorldEntity**(*var*, *name=None*, *desc=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 **name of the property.** (*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

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

Return type List[*Proposition*]

id
Unique name used internally.

Return type str

properties
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*]

type
Type of this entity.

Return type str

class textworld.generator.maker.**WorldPath**(*src*, *src_exit*, *dest*, *dest_exit*, *door=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.

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

Return type Optional[*WorldEntity*]

facts
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: *textworld.generator.maker.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

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*)

8.4 Grammar

class textworld.generator.text_generation.**CountOrderedDict**
Bases: collections.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_infos*)

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_infos*)

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_infos*)

Return a cleaned/keyword replaced symbol.

textworld.generator.text_generation.**generate_instruction**(*action*, *grammar*, *game_infos*, *world*, *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_infos*)
Reduce the action list by combining similar actions.

textworld.generator.text_generation.**is_seq**(*chain*, *game_infos*)
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_infos*, *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.

class textworld.generator.text_grammar.**Grammar**(*options={}*, *rng=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 'l'.
- **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

classmethod deserialize(*data*)
Creates a *GrammarOptions* from serialized data.

Parameters **data** (Mapping[~KT, +VT_co]) – 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[~KT, +VT_co]

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

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

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

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

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

names_to_exclude
List[str] – List of names the text generation should not use.

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

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

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

uuid
Generate UUID for this set of grammar options.

Return type `str`

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

8.5 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*)

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.

8.5.1 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 {
            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}";
```

(continues on next page)

(continued from previous page)

```
            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 {
        lock/d :: unlock/d;
        open/d :: close/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();
```

(continues on next page)

(continued from previous page)

```
        # 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}";

            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();
```

(continues on next page)

(continued from previous page)

```
        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 {
        take :: drop;
        take/c :: insert;
        take/s :: put;
    }

    inform7 {
        predicates {
```

(continues on next page)

(continued from previous page)

```
            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}";

            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();
        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.
    }

    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);
```

(continues on next page)

(continued from previous page)

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

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

(continues on next page)

(continued from previous page)

```
            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);
    }

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

(continues on next page)

(continued from previous page)

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

8.6 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

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 9

textworld.challenges

9.1 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.

`textworld.challenges.coin_collector.`**`build_argparser`**(*parser=None*)

`textworld.challenges.coin_collector.`**`make`**(*settings*, *options=None*)

Make a Coin Collector game of the desired difficulty settings.

Parameters

- **`settings`** (`Mapping[str, str]`) – Difficulty level (see notes). Expected pattern: level[1-300].
- **`options`** (`Optional[`*`GameOptions`*`]`) – For customizing the game generation (see *`textworld.GameOptions`* for the list of available options).

Return type *`Game`*

Returns Generated game.

Notes

Difficulty levels are defined as follows:

- Level 1 to 100: Nb. rooms = level, quest length = level
- Level 101 to 200: Nb. rooms = 2 * (level % 100), quest length = level % 100, distractors rooms added along the chain.
- Level 201 to 300: Nb. rooms = 3 * (level % 100), quest length = level % 100, distractors rooms *randomly* added along the chain.
- ...

textworld.challenges.coin_collector.**make_game**(*mode*, *options*)
Make a Coin Collector game.

Parameters

- **mode** (str) – Mode for the game where
  - 'simple': the distractor rooms are only placed orthogonaly to the chain. This means moving off the optimal path leads immediately to a dead end.
  - 'random': the distractor rooms are randomly place along the chain. This means a player can wander for a while before reaching a dead end.
- **options** (*GameOptions*) – For customizing the game generation (see *textworld.GameOptions* for the list of available options).

Return type *Game*

Returns Generated game.

9.2 Treasure Hunter

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

References

textworld.challenges.treasure_hunter.**build_argparser**(*parser=None*)

textworld.challenges.treasure_hunter.**make**(*settings*, *options=None*)
Make a Treasure Hunter game of the desired difficulty settings.

Parameters

- **settings** (Mapping[str, str]) – Difficulty level (see notes). Expected pattern: level[1-30].
- **options** (Optional[*GameOptions*]) – For customizing the game generation (see *textworld.GameOptions* for the list of available options).

Returns Generated game.

Return type game

Notes

Difficulty levels are defined as follows:

- Level 1 to 10: mode easy, nb. rooms = 5, quest length ranging from 1 to 5 as the difficulty increases;
- Level 11 to 20: mode medium, nb. rooms = 10, quest length ranging from 2 to 10 as the difficulty increases;
- Level 21 to 30: mode hard, nb. rooms = 20, quest length ranging from 3 to 20 as the difficulty increases;

where the different modes correspond to:

- Easy: rooms are all empty except where the two objects are placed. Also, connections between rooms have no door.
- Medium: adding closed doors and containers that might need to be open in order to find the object.

- Hard: adding locked doors and containers (necessary keys will in the inventory) that might need to be unlocked (and open) in order to find the object.

textworld.challenges.treasure_hunter.**make_game**(*mode*, *options*)
Make a Treasure Hunter game.

Parameters

- **mode** (str) – Mode for the game where
  - 'easy': rooms are all empty except where the two objects are placed. Also, connections between rooms have no door.
  - 'medium': adding closed doors and containers that might need to be open in order to find the object.
  - 'hard': adding locked doors and containers (necessary keys will in the inventory) that might need to be unlocked (and open) in order to find the object.
- **options** (*GameOptions*) – For customizing the game generation (see *textworld.GameOptions* for the list of available options).

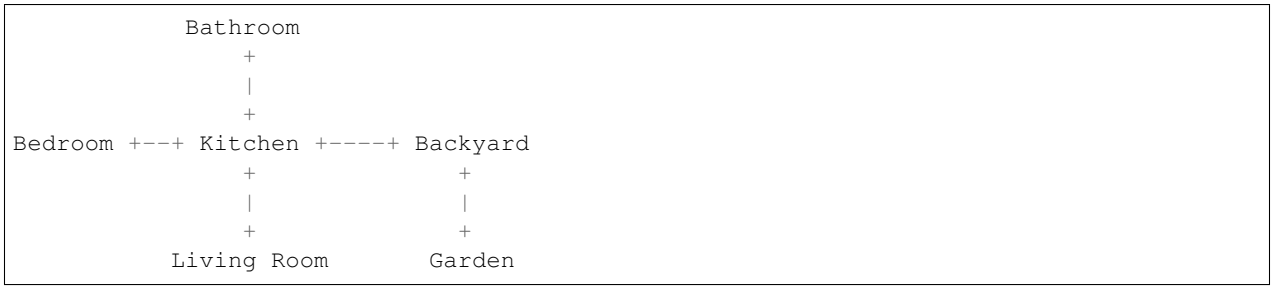
Return type *Game*

Returns Generated game.

9.3 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.



textworld.challenges.simple.**build_argparser**(*parser=None*)

textworld.challenges.simple.**make_game**(*settings*, *options=None*)
Make a simple game.

Parameters

- **settings** (Mapping[str, str]) – Difficulty settings (see notes).
- **options** (Optional[*GameOptions*]) – For customizing the game generation (see *textworld.GameOptions* for the list of available options).

Return type *Game*

Returns Generated game.

Notes

The settings that can be provided are:

- rewards : The reward frequency: dense, balanced, or sparse.
- goal : 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.

CHAPTER 10

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*

inverse(*name=None*)

Invert the direction of this action.

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

Returns

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[~KT, +VT_co]

added
All the new propositions being introduced by this action.

Return type Collection[*Proposition*]

all_propositions
All the pre- and post-conditions.

Return type Collection[*Proposition*]

removed
All the old propositions being removed by this action.

Return type Collection[*Proposition*]

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[~KT, +VT_co]

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.

Returns

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`[~KT, +VT_co]

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*

names
The names of the placeholders in this predicate.

Return type `Collection`[`str`]

types
The types of the placeholders in this predicate.

Return type `Collection`[`str`]

class textworld.logic.**Proposition**(*name*, *arguments=[]*)
Bases: `object`

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`[~KT, +VT_co]

arguments

name

names
The names of the variables in this proposition.

Return type Collection[str]

signature

types
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.

Returns

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.

Returns

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[~KT, +VT_co]

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*

all_predicates
All the pre- and post-conditions.

Return type Iterable[*Predicate*]

class textworld.logic.**Signature**(*name*, *types*)
Bases: object

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**(*facts=None*)
Bases: object

The current state of a world.

Create a State.

Parameters **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.

Returns

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.

Returns

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.

Returns

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[+T_co]

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*]

facts
All the facts in the current state.

Return type Iterable[*Proposition*]

variables
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

ancestors
The ancestors of this type (not including itself).

Return type Iterable[*Type*]

child_types
The direct children of this type.

Return type Iterable[*Type*]

children
The names of the direct children of this type.

Return type Iterable[str]

descendants
The descendants of this type (not including itself).

Return type Iterable[*Type*]

parent_types
The parents of this type as Type objects.

Return type Iterable[*Type*]

subtypes
This type and its descendants.

Return type Iterable[*Type*]

supertypes
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*)

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:

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

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

```
            __ (A,A) __
          /    |    |   \
      (A,B) (A,C) (B,A) (C,A)
  ********************************
(A,D) (B,B) (B,C) (C,B) (C,C) (D,A)
  ********************************
      (B,D) (C,D) (D,B) (D,C)
           \    |    |    /
            \_ (D,D) _/
```

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*]]

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[~KT, +VT_co]

name

type

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

Bases: *textworld.logic.model.ModelBase*

name = None

postconditions = None

preconditions = None

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

Bases: *textworld.logic.model.ModelBase*

condition = None

preserve = None

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

Bases: *textworld.logic.model.ModelBase*

lhs = None

rhs = None

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

Bases: *textworld.logic.model.ModelBase*

constraints = None

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

Bases: *textworld.logic.model.ModelBase*

types = None

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

Bases: tatsu.semantics.ModelBuilderSemantics

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

Bases: *textworld.logic.model.ModelBase*

code = None

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

Bases: *textworld.logic.model.ModelBase*

command = None

event = None

rule = None

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

Bases: *textworld.logic.model.ModelBase*

commands = None

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

Bases: *textworld.logic.model.ModelBase*

parts = None

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

Bases: *textworld.logic.model.ModelBase*

predicate = None

source = None

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

Bases: *textworld.logic.model.ModelBase*

predicates = None

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

Bases: *textworld.logic.model.ModelBase*

definition = None

kind = None

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

Bases: tatsu.objectmodel.Node

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

Bases: *textworld.logic.model.ModelBase*

name = None

type = None

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

Bases: *textworld.logic.model.ModelBase*

name = None

parameters = None

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

Bases: *textworld.logic.model.ModelBase*

predicates = None

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

Bases: *textworld.logic.model.ModelBase*

arguments = None

name = None

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

Bases: *textworld.logic.model.ModelBase*

lhs = None

rhs = None

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

Bases: *textworld.logic.model.ModelBase*

reverse_rules = None

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

Bases: *textworld.logic.model.ModelBase*

name = None

postconditions = None

preconditions = None

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

Bases: *textworld.logic.model.ModelBase*

condition = None

preserve = None

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

Bases: *textworld.logic.model.ModelBase*

rules = None

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

Bases: *textworld.logic.model.ModelBase*

name = None

types = None

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

Bases: *textworld.logic.model.ModelBase*

name = None

parts = None

supertypes = None

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

Bases: *textworld.logic.model.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: tatsu.buffering.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: tatsu.parsing.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 11

textworld.render

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()

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.**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** (*GlulxGameState*) – 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*, *GlulxGameState*, *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.

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: <built-in function 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: *GlulxGameState* :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 12

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*)

seed

class textworld.utils.**RegexDict**
Bases: collections.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.**chunk**(*iterable*, *n*, *fct=<function <lambda>>*)

Return type Iterable[Iterable[+T_co]]

textworld.utils.**encode_seeds**(*seeds*)
Generate UID from a list of seeds.

textworld.utils.**get_webdriver**(*path=None*)
Get the driver and options objects. :param path: path to browser binary. :return: driver

textworld.utils.**make_temp_directory**(*suffix=''*, *prefix='tmp'*, *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[+T_co]

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.**which**(*program*)
helper to see if a program is in PATH :param program: name of program :return: path of program or None

textworld.utils.**g_rng = <textworld.utils.RandomGenerator object>**
Global random generator.

CHAPTER 13

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

Z