```
import gym
import pddlgym.structs as pgym
import pyperplan.pddl.pddl as pyper
import collections

"""
Bunch of wrappers useful to convert between PDDLGym and other representations
"""

# String Wrapper
class StringWrapper(gym.Wrapper):
    """
    Allows for providing actions as strings rather than the pddlgym representation
    """
    def __init__(self, env):
        super().__init__(env)
        self.env = env

    def step(self,action_str):
        action = self._str_to_action(action_str)
        print(f"Action: {action}", type(action))
        next_state, reward, done, info = self.env.step(action)
        return next_state, reward, done, info

    def _str_to_action(self, action_str):
        name = action_str.split("(")[0]
        args = action_str.split("(")[1].split(")")[0].split(",")
        objs = []
        for arg in args:
            arg = arg.replace(" ","")
            obj_type_str = arg.split(":")[1]
            obj_const_str = arg.split(":")[0]

            obj_type = pgym.Type(obj_type_str)
            obj_const = obj_type(obj_const_str)
            objs.append(obj_const)
        actionP = pgym.Predicate(name, len(args))
        action = pgym.Literal(actionP, objs)
        return action

# pyperplan wrapper
class PyperWrapper(gym.Wrapper):
    """
    Allows for translating between pddlgym and pyperplan representations.
    Can run pyperplan solutions in pddlgym
    Can read pddlgym output in pyperplan state representation
    """
    def __init__(self, env):
        super().__init__(env)
        self.env = env
        self.current_state = None  # going to be in default pddlgym representation
        self.root_object_type = pyper.Type("object",None)
        self.history = [] #In pyperplan format

    def reset(self, **kwargs):
        obs = self.env.reset(**kwargs)
        self.current_state = obs
        self.type_keyed_objects, self.token_keyed_objects = self._get_objects() # ob
jects from current state
        return self.observe()

    def observe(self, source='pypertask'):
        if source == 'pyperpred':
            return self._obs_pddlgym_to_pyper(self.current_state)
        if source == 'pddlgym':
            return self.current_state
        if source == 'pypertask':
            return self._obs_pddlgym_to_pypertask(self.current_state)
        raise ValueError("source must be either 'pyper' or 'pddlgym'")

    def objects(self):
        """
        Returns all the objects in the current state
        """
        objects = {}
        objs = self.current_state[0].objects
        for o in set(objs):
            key = o.name
            type_name = str(o.var_type)
            value = pyper.Type(type_name, self.root_object_type)
            objects.update({key : value})
        return objects



    def step(self, action):
        print("\t\t\t===DEBUG===")
        print(f"\t\t\tAction submitted by agent: {action}")
        action = self._ground_literal(action) #Allows me to provide lifted actions
        print(f"\t\t\tGrounded Action: {action}")
        action_pddlgym = self._action_pyper_to_pddlgym(action)
        print(f"\t\t\tActual PDDLGym Action: {action_pddlgym}")
        next_state = self.env.step(action_pddlgym)
        self.history.append(action)
        self.current_state = next_state
        # print(f"\t\t\tCurrent state: {self.current_state}")

        return self.observe()

##############


    def _ground_literal(self, literal):
        """
        returns a literal grounded in either objects the agent knows about OR is hyp
othesized
        """
        if self._is_grounded_literal(literal):
            return literal

        name, args = self._parse_literal(literal)
        grounded_args = []
        for arg in args:
            grounded_arg = self._ground_arg(arg)
            grounded_args.append(grounded_arg)

        return self._construct_literal(name, grounded_args)


    def _construct_literal(self, name, args):
        """
        Returns a literal based on name and args
        (have t23)
        """
        return f"({name} {' '.join(args)})"


    def _ground_arg(self, arg):
        """
        Returns a grounded arg
        NOTE: if the object is hypothetical, this is added via self.add_objects()
        """
        if self._is_grounded_arg(arg):
            return arg
        symbol, typing = self._parse_arg(arg)
        if self.type_keyed_objects[typing]:
            symbol = list(self.type_keyed_objects[typing])[0]
            return symbol
        breakpoint()
        raise ValueError("Not allowed to hypothesize objects here")
        return None


    def _parse_arg(self, arg):
        """
        Returns symbol, typing
        input: ?x-t, t23-t, t23, *t234-t
        output: ?x,t  or t23,t or t23,t or *t234-t
        """

        if "?" in arg:
            if "-" in arg:
                symbol = arg.split("-")[0]
                typing = arg.split("-")[1]
                return symbol, typing
            else:
                raise ValueError(f"Argument ({arg}) Must have at least constant or t
ype")
        if "-" in arg:
            symbol = arg.split("-")[0]
            typing = arg.split("-")[1]
            return symbol, typing

        symbol = arg
        try:
            typing = self.token_keyed_objects[symbol]
            return symbol, typing
        except:
            raise ValueError(f"The object {arg} does not exist anywhere")


    def _is_grounded_literal(self, literal):
        """
        Returns true of literal is grounded
        """

        name, args = self._parse_literal(literal)
        for arg in args:
            if not self._is_grounded_arg(arg):
                return False
        return True


    def _parse_literal(self, literal):
        """
        Gets name, and arguments from a string literal as a string, list
        note: an arg could be "t23" or "?x-t" or "t23-t", assuming well formed.
        """

        name = literal.replace("(","").replace(")","").split(" ")[0]
        args = literal.replace("(","").replace(")","").split(" ")[1:]
        return name, args

    def _is_grounded_arg(self, arg):
        """
        Given an arg, checks if it is grounded
        arg = "?x-t" or "t23" or "t23-t" or "*t234"
        """

        if "?" in arg:
            return False
        return True


    def _get_objects(self):
        """
        Returns objects as a dicts, keyed by types, keyed by object
        """

        type_keyed_objects = collections.defaultdict(set)
        token_keyed_objects = collections.defaultdict()
        objects_dict = self.objects() ###*** LOOKS AT ENV ***********

        for k, v in objects_dict.items():
            type_keyed_objects[v.name].add(k)
            token_keyed_objects[k] = v.name

        return type_keyed_objects, token_keyed_objects
```

```
###########




    def get_history(self):
        return self.history

    def _action_pyper_to_pddlgym(self, action):
        """
        (move p3 k1) --> pddlgym representation
        """
        name = action.replace("(","").replace(")","").split(" ")[0]
        objs = set(self.current_state[0].objects)
        args = action.replace("(","").replace(")","").split(" ")[1:]
        arity = len(args)
        typed_args = [ (lambda x: [x for x in objs if x == y][0])(y) for y in args]
        action_pddlgym = pgym.Literal(pgym.Predicate(name, arity), typed_args)
        return action_pddlgym


    def _obs_pddlgym_to_pyper(self, obs):
        predicates = []
        for item in set(obs[0].literals):
            name = str(item).split("(")[0]
            list_args_str = str(item).split("(")[1].split(")")[0].split(",")
            signature = [(x.split(":")[0], [pyper.Type(x.split(":")[1], self.root_object_type)]) for x in list_args_str]
            pred = pyper.Predicate(name, signature)
            predicates.append(pred)
        return predicates

    def _obs_pddlgym_to_pypertask(self, obs):
        literals = []
        for item in set(obs[0].literals):
            name = str(item).split("(")[0]
            list_args_str = str(item).split("(")[1].split(")")[0].split(",")
            new_args = [x.split(":")[0] for x in list_args_str]
            new_lit = "("+name+" "+" ".join(new_args)+")"
            literals.append(new_lit)
        return frozenset(literals)
```