```
import ply.lex as lex
import ply.yacc as yacc
import sys
import os
sys.path.append(’..’)
from imperative_parser.parser import parse_function
from imperative_parser.utils import find_column

tokens = (
    ’COLON’,
    ’LCURLY’,
    ’RCURLY’,
    ’LBRACKET’,
    ’RBRACKET’,
    ’COMMA’,
    ’DOT’,
    ’WILD’,
    ’PLUS’,
    ’ID’,
    ’EXTENSION’,
    ’STR’,
    ’FUNC’,
    ’NUM’
)

reserved = {
    ’uniform’: ’UNIFORM’,
    ’None’: ’NONE’
}

tokens += tuple(reserved.values())
t_COLON = r’:’
t_LCURLY = r’\{’
t_RCURLY = r’\}’
t_LBRACKET = r’\[’
t_RBRACKET = r’\]’
t_COMMA = r’,’
t_DOT = r’\.’
t_WILD = r’\*’
t_PLUS = r’\+’
t_STR = r’".*"’
t_ignore = ’ \t’

def t_FUNC(t):
    r’func[^\{]*{’
    func = t.value
    bracks = 1
    pos = t.lexer.lexpos
    pos2 = t.lexpos
    lexdata = t.lexer.lexdata[t.lexer.lexpos:]
    for c in lexdata:
        t.lexer.lexpos += 1
        func += c
        if c == ’{’:
            bracks += 1
        elif c == ’}’:
            bracks -= 1
        elif c == ’\n’:
            t.lexer.lineno += 1
        if not bracks:
            break
    t.value = func
    return t

def t_ID(t):
    r’[A-Za-z][A-Za-z-]*’
    t.type = reserved.get(t.value, ’ID’)
    return t

def t_EXTENSION(t):
    r’@extend’
    return t

def t_NUM(t):
    r’\d+’
    t.value = int(t.value)
    return t

def t_newline(t):
    r’\n+’
    t.lexer.lineno += len(t.value)

def t_error(t):
    print "Illegal character ’%s’" % t.value[0]

lexer = lex.lex()

# Error Handling
SUCCEEDED = True
PARSED_STRING = ""

def p_property_value(p):
    ’property : ID COLON value’
    p[0] = {p[1]: p[3]}

def p_property_extension(p):
    ’property : EXTENSION COLON value’
    p[0] = {p[1]: p[3]}

def p_value_structure(p):
    ’value : structure’
    p[0] = p[1]

def p_value_list(p):
    ’value : LBRACKET list RBRACKET’
    p[0] = p[2]

def p_value_dots(p):
    ’value : dots’
    p[0] = p[1]

def p_value_num(p):
    ’value : NUM’
    p[0] = p[1]

def p_value_str(p):
    ’value : STR’
    p[0] = p[1].strip(’\’"’)

def p_value_uniform(p):
    ’value : UNIFORM’
    p[0] = ’uniform’

def p_value_none(p):
    ’value : NONE’
    p[0] = None

def p_value_func(p):
    ’value : FUNC’
    global SUCCEEDED
    try:
        p[0] = parse_function(p[1], line_offset=p.lineno(1), col_offset=find_column(
PARSED_STRING, lexpos=p.lexpos(1)))
    except:
        SUCCEEDED = False
        p[0] = p[1]

def p_structure_properties(p):
    ’structure : LCURLY properties RCURLY’
    p[0] = p[2]

def p_list_comma(p):
    ’list : value COMMA list’
    p[1] = [p[1]]
    p[1].extend(p[3])
    p[0] = p[1]

def p_list_value(p):
    ’list : value’
    p[0] = [p[1]]

def p_dots_dot(p):
    ’dots : ID DOT dots’
    p[0] = p[1] + ’.’ + p[3]

def p_dots_plus(p):
    ’dots : ID PLUS NUM’
    p[0] = p[1] + ’ + ’ + str(p[3])

def p_dots_id(p):
    ’dots : ID’
    p[0] = p[1]

def p_dots_wild(p):
    ’dots : WILD’
    p[0] = ’*’

def p_properties_comma(p):
    ’properties : property COMMA properties’
    p[3].update(p[1])
    p[0] = p[3]

def p_properties_property(p):
    ’properties : property’
    p[0] = p[1]

def p_error(p):
    print p
    print "Syntax error in input!"

parser = yacc.yacc()

def parse(s):
    global SUCCEEDED
    global PARSED_STRING
    PARSED_STRING = s
    return parser.parse(s, lexer=lexer), SUCCEEDED
```

```

# parsetab.py
# This file is automatically generated. Do not edit.
_tabversion = ’3.2’

_lr_method = ’LALR’

_lr_signature = ’)\xbc4\x8d\x11K\x81t\x9c\xb8\x8f8\xdb\x0f_\xf3’

_lr_action_items = {’PLUS’:([16,],[23,]),’NONE’:([4,5,11,26,],[7,7,7,7,]),’FUNC’:([4,5,11,26,],[8,8,8,8,]),’EXTENSION’:([0,12,28,],[3,3,3,]),’RCURLY’:([6,7,8,9,10,13,14,15,16,17,18,21,22,25,27,29,30,32,],[-5,-9,-10,-1,-8,-6,-7,-17,-16,-3,-2,27,-19,-4,-11,-15,-14,-18,]),’UNIFORM’:([4,5,11,26,],[10,10,10,10,]),’LBRACKET’:([4,5,11,26,],[11,11,11,11,]),’LCURLY’:([4,5,11,26,],[12,12,12,12,]),’NUM’:([4,5,11,23,26,],[13,13,13,29,13,]),’COLON’:([2,3,],[4,5,]),’STR’:([4,5,11,26,],[14,14,14,14,]),’WILD’:([4,5,11,24,26,],[15,15,15,15,15,]),’COMMA’:([6,7,8,9,10,13,14,15,16,17,18,20,22,25,27,29,30,],[-5,-9,-10,-1,-8,-6,-7,-17,-16,-3,-2,26,28,-4,-11,-15,-14,]),’RBRACKET’:([6,7,8,10,13,14,15,16,17,19,20,25,27,29,30,31,],[-5,-9,-10,-8,-6,-7,-17,-16,-3,25,-13,-4,-11,-15,-14,-12,]),’ID’:([0,4,5,11,12,24,26,28,],[2,16,16,16,2,16,16,2,]),’DOT’:([16,],[24,]),’$end’:([1,6,7,8,9,10,13,14,15,16,17,18,25,27,29,30,],[0,-5,-9,-10,-1,-8,-6,-7,-17,-16,-3,-2,-4,-11,-15,-14,]),}

_lr_action = { }
for _k, _v in _lr_action_items.items():
   for _x,_y in zip(_v[0],_v[1]):
      if not _x in _lr_action:  _lr_action[_x] = { }
      _lr_action[_x][_k] = _y
del _lr_action_items

_lr_goto_items = {’dots’:([4,5,11,24,26,],[6,6,6,30,6,]),’list’:([11,26,],[19,31,]),’value’:([4,5,11,26,],[9,18,20,20,]),’property’:([0,12,28,],[1,22,22,]),’properties’:([12,28,],[21,32,]),’structure’:([4,5,11,26,],[17,17,17,17,]),}

_lr_goto = { }
for _k, _v in _lr_goto_items.items():
   for _x,_y in zip(_v[0],_v[1]):
       if not _x in _lr_goto: _lr_goto[_x] = { }
       _lr_goto[_x][_k] = _y
del _lr_goto_items
_lr_productions = [
  ("S’ -> property","S’",1,None,None,None),
  (’property -> ID COLON value’,’property’,3,’p_property_value’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,93),
  (’property -> EXTENSION COLON value’,’property’,3,’p_property_extension’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,97),
  (’value -> structure’,’value’,1,’p_value_structure’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,101),
  (’value -> LBRACKET list RBRACKET’,’value’,3,’p_value_list’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,105),
  (’value -> dots’,’value’,1,’p_value_dots’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,109),
  (’value -> NUM’,’value’,1,’p_value_num’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,113),
  (’value -> STR’,’value’,1,’p_value_str’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,117),
  (’value -> UNIFORM’,’value’,1,’p_value_uniform’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,121),
  (’value -> NONE’,’value’,1,’p_value_none’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,125),
  (’value -> FUNC’,’value’,1,’p_value_func’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,129),
  (’structure -> LCURLY properties RCURLY’,’structure’,3,’p_structure_properties’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,138),
  (’list -> value COMMA list’,’list’,3,’p_list_comma’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,142),
  (’list -> value’,’list’,1,’p_list_value’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,148),
  (’dots -> ID DOT dots’,’dots’,3,’p_dots_dot’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,152),
  (’dots -> ID PLUS NUM’,’dots’,3,’p_dots_plus’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,156),
  (’dots -> ID’,’dots’,1,’p_dots_id’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,160),
  (’dots -> WILD’,’dots’,1,’p_dots_wild’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,164),
  (’properties -> property COMMA properties’,’properties’,3,’p_properties_comma’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,168),
  (’properties -> property’,’properties’,1,’p_properties_property’,’/Users/mdzhang/Projects/pltcatan/config_parser/config.py’,173),
]
```

```
#!/usr/bin/env python
import config
import argparse
import dill as pickle
import os
import shutil
import sys
sys.path.append(’..’)
from engine.src.game import Game
from engine.src.config.config import Config

properties = {}

def undot(property):
    ’’’
    Get the value of a dot.notated.property from the properties dict
    ’’’
    extended = properties
    extension = property.split(’.’)
    extension.reverse()
    while extension:
        extended = extended.get(extension.pop(), properties)
        if extended is properties:
            return extended
    if isinstance(extended, dict) or isinstance(extended, list):
        return extended.copy()
    else:
        return extended

def extend_verbose(skit, property, value, extension):
    ’’’
    Extend properties using the verbose syntax where every extension must use an
    @extend explicitly
    ’’’
    skit[property] = undot(extension)
    for extended_property, extended_value in value.iteritems():
        if extended_property != ’@extend’:
            if isinstance(extended_value, str) and ’+’ in extended_value:
                extension, addition = extended_value.split(’+’)
                extended = undot(extension.strip())
                extended_value = extended + int(addition)
            skit[property][extended_property] = extended_value

def extend_clean(skit, property, value, extension):
    ’’’
    Extend properties using the cleaner syntax where one mention of @extend and
    explicit-overwrite-only set to true cascades the extension gracefully
    ’’’
    explicit = extension[’explicit-overwrite-only’]
    extension = extension[’value’]
    extend_verbose(skit, property, value, extension)
    if explicit:
        for extended_property, extended_value in value.iteritems():
            if isinstance(extended_value, dict) and extended_property !=\
                    ’@extend’:
                if needs_extending(extended_value):
                    skit[property][extended_property][’@extend’] =\
                            make_extend(extension, extended_property, explicit)
    return extension

def needs_extending(skit):
    ’’’
    Checks to see if a structure needs to be extended
    ’’’
    children_structures = False
    if isinstance(skit, dict):
        return True
    for property, value in skit.iteritems():
        if isinstance(value, dict):
            children_structures = True
    return children_structures

def make_extend(extension, extended_property, explicit):
    ’’’
    Coerce the structure to look like a verbose extension
    ’’’
    return {’value’: ’%s.%s’ % (extension, extended_property),
            ’explicit-overwrite-only’: explicit}

def replace(value):
    ’’’
    Replace an import alias with its actual value
    ’’’
    if ’+’ in value:
        terms = value.split(’+’)
        sum = 0
        for term in terms:
            term = term.strip()
            if term.isdigit():
                replacement = float(term)
            else:
                replacement = undot(term.strip())
                if replacement is properties:
                    sum = None
                    break
            sum += float(replacement)
        if sum is None:
            replacement = value
        else:
            replacement = sum
    else:
        replacement = undot(value.strip())
    if replacement is properties:
        return value
    else:
        return replacement

def extend(skit, parent=None):
    ’’’
    Replace all extended properties with the contents of the actual value
    denoted by the dot-notated property name and set any additional properties
    ’’’
    for property, value in skit.iteritems():
        if isinstance(value, str):
            replacement = replace(value)
            if isinstance(replacement, dict):
                replacement = replacement.get(property, replacement)
            skit[property] = replacement
        if isinstance(value, dict):
            extension = value.get(’@extend’)
            if extension:
                if isinstance(extension, str):
                    extend_verbose(skit, property, value, extension)
                    extension = None
                else:
                    extension = extend_clean(skit, property, value, extension)
            extend(skit[property])

def imports(full_file, file):
    ’’’
    Compiles every skit structure that is imported in addition to
    the top-level structure
    ’’’
    imports = file.split(’\n’)
    line_no = 0
    chars_read = 0
    for line in imports:
        line_length = len(line)
        if line:
            line = line.split()
            if line[0] == ’@import’:
                if len(line) < 4:
                    print ’Error: Invalid @import on line’, line_no
                    return None
                if line[1][-1] == ’/’:
                    if line[1][0] == ’.’:
                        properties[line[3]], success = compile(full_file + line[1] + \
                                ’__value__.skit’, as_name=line[3])
                    elif line[1][0] == ’/’:
                        properties[line[3]], success = compile(line[1] +\
                                ’__value__.skit’, as_name=line[3])
                else:
                    if line[1][0] == ’.’:
                        properties[line[3]], success = compile(full_file + line[1] + \
                                ’.skit’)
                    elif line[1][0] == ’/’:
                        properties[line[3]], success = compile(line[1] + ’.skit’)

            else:
                break
        line_no += 1
        chars_read += line_length
    if chars_read > 0:
        chars_read += 1
    return file[chars_read:]

def compile(file, clean=False, as_name=None):
    ’’’
    Cleans tmp/ directory and reinitializes with compiled skit code
    ’’’
    full_file = os.path.dirname(file) + ’/’
    base_file = os.path.basename(file)
    compile_file = ’tmp/’ + base_file
    if clean:
        shutil.rmtree(’tmp/’, True)
        compile(’default.skit’)
    file = open(file, ’r’).read()
    file = imports(full_file, file)
    skit, succeeded = config.parse(file)
    main_property = os.path.splitext(base_file)[0]
    extend(skit)
    if as_name:
        properties[as_name] = skit
        main_property = as_name
    else:
        properties[main_property] = skit.get(main_property)
    if not os.path.isdir(’tmp/’):
        os.makedirs(’tmp/’)
    pickle.dump(skit, open(compile_file, ’wb’))
    return skit, succeeded

def run(file):
    ’’’
    Runs skit game
    Recompiles skit code only if code has been changed
    ’’’
    _, success = compile(’default.skit’)
    if success:
        base_file = os.path.basename(file)
        compile_file = ’tmp/’ + base_file
        skit = None
        if not os.path.isfile(compile_file) or\
            os.path.getmtime(file) > os.path.getmtime(compile_file):
            skit = compile(file)[0]
        else:
            skit = pickle.load(open(compile_file, ’rb’))
        main_property = os.path.splitext(base_file)[0]
        properties[main_property] = skit.get(main_property)
        Config.config = properties[main_property]
        Config.init()
        game = Game()
        skit = skit.get(os.path.splitext(base_file)[0], None)
        # TODO: restore after engine syncs config dict format
        # if skit.get(’game’, None):
        game.start()
    else:
        print "Build failed, check the log for errors"
        sys.exit(1)

if __name__ == ’__main__’:
    arg_parser = argparse.ArgumentParser(description=’Skit compiler’)
    arg_parser.add_argument(’file’, help=’Skit file’)
    arg_parser.add_argument(’-c’, ’--compile’, action=’store_true’,
        help=’Only run compile steps’)
    args = arg_parser.parse_args()
    if args.compile:
        compile(args.file, True)
    else:
        run(args.file)
```

```python
#!/usr/bin/env python
import sys
import config
import skit

passed_all = True

def dummy():
    return 0

recognized_types = [type(''), type(0), type(dict()), type(list()), type(None),\
        type(dummy)]
function_names = ['play-card', 'draw-card']
string_names = ['name', 'description', 'position-type']
int_names = ['points-to-win', 'player-count', 'radius', 'tile-count', 'count',\
        'point-value', 'base-yield']
structure_names = ['game', 'board', 'card', 'development', 'structure',\
        'player-built']

def type_per_name(skit, property, value):
    can_be_none = False
    global passed_all
    if property in function_names:
        if type(value) != type(dummy):
            print 'Error: property %s does not contain a function' % property
            print 'Actual type: %s', type(value)
            passed_all = False
    elif property in string_names:
        if type(value) != type(''):
            print 'Error: property %s does not contain a string' % property
            print 'Actual type: %s', type(value)
            passed_all = False
    elif property in int_names:
        if type(value) != type(0):
            print 'Error: property %s does not contain an integer' % property
            print 'Actual type: %s', type(value)
            passed_all = False
    elif property in structure_names:
        if type(value) != type(dict()):
            print 'Error: property %s does not contain a dict' % property
            print 'Actual type: %s', type(value)
            passed_all = False

def test_types(skit):
    if type(skit) not in recognized_types:
        print 'Error: %s has unrecognized type', (skit, type(skit))
    if isinstance(skit, dict):
        for property, value in skit.iteritems():
            if property == 'default' and not skit[property].get('game', None):
                continue
            else:
                type_per_name(skit, property, value)
            test_types(value)

if __name__ == '__main__':
    game = config.parser.parse(open('default.skit', 'r').read())
    default = skit.compile('default.skit')
    test_types(game)
    test_dict = {'test': {'game': {'points-to-win': 5 } } }
    test_skit = 'test: { game: { points-to-win: 5 } }'
    compiled_skit = config.parser.parse(test_skit)
    if test_dict != compiled_skit:
        print 'Error: Static test dict does not match compiled test.skit'
        print 'Static test dict: %s', test_dict
        print 'Compiled test.skit: %s', compiled_skit
        passed_all = False
    test_dict['test']['game']['points-to-win'] = 10
    if test_dict == compiled_skit:
        print 'Error: Static test dict matches compiled test.skit with lower \
points to win'
        print 'Static test dict: %s', test_dict
        print 'Compiled test.skit: %s', compiled_skit
        passed_all = False
    test_skit = 'test: { game: { points-to-win: default.game.points-to-win } }'
    compiled_skit = config.parser.parse(test_skit)
    skit.extend(compiled_skit)
    if test_dict != compiled_skit:
        print 'Error: Static test dict does not match compiled test.skit\'s \
points-to-win'
        print 'Static test dict: %s', test_dict
        print 'Compiled test.skit: %s', compiled_skit
        passed_all = False
    test_skit = 'test: { game: default.game }'
    first_compile = config.parser.parse(test_skit)
    second_compile = config.parser.parse(test_skit)
    if first_compile != second_compile:
        print 'Error: Equivalent skit structures do not match when compiled'
        print 'Static test dict: %s', test_dict
        print 'Compiled test.skit: %s', compiled_skit
        passed_all = False
    skit.extend(first_compile)
    skit.extend(second_compile)
    if first_compile != second_compile:
        print 'Error: Equivalent skit structures do not match when extended'
        print 'Static test dict: %s', test_dict
        print 'Compiled test.skit: %s', compiled_skit
        passed_all = False
    first_skit = 'skit: { one: { a: 5, b: 6, c: 4 }, two: { b: 6, a: 5, c: 4 } }'
    second_skit = 'skit: { two: { b: 6, a: 5, c: 4 }, one: { a: 5, b: 6, c: 4 } }'
    first_compile = config.parser.parse(first_skit)
    second_compile = config.parser.parse(second_skit)
    if first_compile != second_compile:
        print 'Error: Semantically skit structures do not match when extended'
        print 'Static test dict: %s', test_dict
        print 'Compiled test.skit: %s', compiled_skit
        passed_all = False
    if passed_all:
        print 'Passed every test!'
```

```
# makefile

start: start.py
        python start.py

debug:
        # pdb.set_trace()
        python -m pdb start.py

.PHONY: clean
clean:
        find . -name "*.pyc" -exec rm -rf {} \;
```

```python
# -*- coding: utf-8 -*-


class Board(object):
    pass
```

```
# -*- coding: utf-8 -*-
import random
import pdb

from engine.src.lib.utils import Utils
from engine.src.board.hex_board import HexBoard
from engine.src.tile.game_tile import GameTile
from engine.src.resource_type import ResourceType
from engine.src.position_type import PositionType
from engine.src.calamity.calamity import Calamity
from engine.src.calamity.calamity import CalamityTilePlacementEffect
from engine.src.calamity.robber import Robber
from engine.src.trading.bank import Bank
from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping
from engine.src.direction.edge_direction import EdgeDirection
from engine.src.direction.vertex_direction import VertexDirection
from engine.src.exceptions import *
from engine.src.structure.structure import Structure


class GameBoard(HexBoard):
    """A Settlers of Catan playing board.

    Attributes:
        radius (int): See HexBoard.

        tiles (dict): See HexBoard.

        tile_cls (class): See HexBoard.

        bank (Bank): Bank of resources the board will interact with.

    Args:
        radius (int): See HexBoard.
    """

    def __init__(self, radius):

        super(GameBoard, self).__init__(radius, GameTile)

        # We have tiles, but they currently have no value and are all FALLOW.
        # Here we assign resource types and chit values.
        self.assign_tile_resources()
        self.assign_tile_chit_values()
        self.assign_tile_harbors()

        self.bank = Bank(len(list(self.iter_tiles())))

    def assign_tile_resources(self, assignment_func=None):
        """Assign resource types to this board’s tiles.

        Args:
            assignment_func (func): Resources will assigned according to this
              function. If not provided, will default to
              self._default_assign_tile_resources()

        Returns:
            None.
        """

        if assignment_func is None:
            self._default_assign_tile_resources()
        else:
            assignment_func()

    def _default_assign_tile_resources(self):
        """Distributes non-fallow resource types across the board evenly.

        Specifically, assigns one ResourceType.FALLOW tile, then splits the
        resource types of the remaining tiles evenly.

        Returns:
            None.

        TODO: Defaults to only one FALLOW tile regardless of board size.
              Perhaps should make fallow tile count relative to board size.
        """

        # Get a randomized list of the tiles of this board.
        tiles = list(self.iter_tiles())
        random.shuffle(tiles)

        resource_type_count = len(ResourceType.get_arable_types())

        # We’ll allocate one fallow tile so divide arable resources among
        # total number of tiles - 1.
        per_resource_count = (len(tiles) - 1) / float(resource_type_count)

        # Say that we find that we need to allocate 3.6 tiles per resource.
        # Clearly we can only allocate whole number tiles. So we take the
        # difference between what we calculated and its floor (e.g. .6),
        # and multiply it by the number of tiles to get the number of
        # leftover tiles that need to be assigned.
        leftover_count = int((per_resource_count - int(per_resource_count)) *
                             resource_type_count)

        per_resource_count = int(per_resource_count)

        # Get a list containing resource_type_count occurrences of each
        # resource_type.
        resources = Utils.flatten(map(
            lambda resource: [resource] * per_resource_count,
            ResourceType.get_arable_types()
        ))

        # We then allocate leftover tiles according to some priority. In a
        # base Settlers of Catan game, this priority manifests as having only
        # 3 brick and ore tiles, by 4 lumber, wool, and wheat tiles.
        while leftover_count:
            resources.append(
                ResourceType.get_priority_arable_types()[leftover_count - 1])
            leftover_count -= 1

        # Add a single occurrence of ResourceType.FALLOW.
        resources.append(ResourceType.FALLOW)

        # Assign the resource types to the shuffled tiles.
        for tile, resource_type in zip(tiles, resources):
            tile.resource_type = resource_type

    def _randomly_assign_tile_resources(self):
        """Randomly assign resource types to this board’s tiles.

        Note that this randomly draws from all ResourceType’s, i.e. including
        ResourceType.FALLOW.

        Returns:
            None.
        """

        for tile in self.iter_tiles():
            tile.resource_type = ResourceType.random()
```

```python

    def assign_tile_chit_values(self, assignment_func=None):
        """Assign chit values to this board’s tiles.

        Args:
            assignment_func (func): Chit values will assigned according to this
              function. If not provided, will default to
              self._default_assign_tile_chit_values()

        Returns:
            None.
        """

        if assignment_func is None:
            self._default_assign_tile_chit_values()
        else:
            assignment_func()

    def _randomly_assign_tile_chit_values(self, start=2, end=12,
                                          exclude=Calamity.DEFAULT_ROLL_VALUES):
        """Randomly assign chit values to this board’s tiles.

        Args:
            start (int): The set of possible chit values from which values to
              assign will be randomly drawn is defined by the range defined by
              start and end.

            end (int): See above.

            exclude (list): A list of values that lie in the range given by
              start and end that should not be included in the set of possible
              chit values.

        Returns:
            None
        """

        chit_values = frozenset(range(start, end + 1)).intersection(exclude)

        for tile in self.iter_tiles():
            tile.chit_value = random.choice(chit_values)

    def _default_assign_tile_chit_values(self, start=2, end=12,
                                         exclude=Calamity.DEFAULT_ROLL_VALUES):
        """Assign chit values in a manner similar to that of the original game.

        Specifically, find out how many times each value would occur if we
        were to distribute them over the board’s non-fallow tiles evenly,
        except for the highest and lowest values (presumably the least likely
        to occur), which should only appear on the board half as often.

        Args:
            start (int): Together with end, defines the range of possible
              chit values.

            end (int): See above.

            exclude (list): A list of values that lie in the range defined by
              start and end that should not be included in the set of possible
              chit values.

        Returns:
            None.

        TODO: Consider storing self.tile_count instead of using the length
              of the iterator. For now, however, performance not an issue.
        """

        chit_values = filter(
            lambda value: value not in exclude, range(start, end + 1)
        )

        min_chit_value = chit_values[0]
        max_chit_value = chit_values[-1]

        # We only want to consider arable tiles.
        arable_tiles = list(self.iter_arable_tiles())
        tile_count = len(arable_tiles)

        # Since the lowest and highest chit values will occur half as
        # frequently, we act as if we were only had len(chit_values) - 1 values.
        per_value_count = tile_count / (len(chit_values) - 1)

        # We want the highest and lowest value chits to appear half as often.
        def get_value_occurrence_count(value):
            if value == min_chit_value or value == max_chit_value:
                return per_value_count / 2
            else:
                return per_value_count

        # Get a list of all the chit values we will place e.g. if we expect
        # to place 5 chits of value 3, then 3 should occur 5 times in the list.
        chit_values_to_assign = Utils.flatten(map(
            lambda value: [value] * get_value_occurrence_count(value),
            chit_values
        ))

        # Assign chit values to arable tiles only.
        for tile, chit_value_to_assign in zip(arable_tiles,
                                              chit_values_to_assign):
            tile.chit_value = chit_value_to_assign

    def assign_tile_harbors(self):
        """Assign harbors to this board.

        TODO: Officially, harbors seem to be placed after every
              3rd then 3rd then 4th edge. This is a pain to program given that
              it only _seems_ that way.
        """

        # TODO
        pass

    def iter_arable_tiles(self):
        """Iterate over this board’s non-fallow i.e. arable tiles."""

        for tile in self.iter_tiles():
            if tile.resource_type != ResourceType.FALLOW:
                yield tile

    def place_vertex_structure(self, x, y, vertex_dir, structure,
                               must_border_claimed_edge=True, struct_x=None,
                               struct_y=None, struct_vertex_dir=None):
        """Place a structure of the given type on the specified vertex.

        Args:
            See self.update_vertex().

            structure (Structure): Structure to replace the specified vertex
              with.

        Returns:
            None.

        Raises:
            InvalidBaseStructureException. If structure to be placed is an
              upgrade or extension of a structure class that hasn’t been
              placed at the defined vertex.
        """

        tile = self.tiles[x][y]
        old_vertex_val = tile.vertices[vertex_dir]

        self.validate_structure_placement(x, y, old_vertex_val, structure,
                                          vertex_dir, must_border_claimed_edge,
                                          struct_x, struct_y, struct_vertex_dir)

        self.update_vertex(x, y, vertex_dir, structure)

    def place_edge_structure(self, x, y, edge_dir, structure,
                             must_border_claimed_edge=True, struct_x=None,
                             struct_y=None, struct_vertex_dir=None):
        tile = self.tiles[x][y]
        vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)
        old_edge_val = tile.edges[vertex_dirs[0]][vertex_dirs[1]]

        self.validate_structure_placement(x, y, old_edge_val, structure,
                                          edge_dir, must_border_claimed_edge,
                                          struct_x, struct_y, struct_vertex_dir)

        self.update_edge(x, y, edge_dir, structure)

    def validate_structure_placement(self, x, y, old_value, new_value,
                                     placement_dir, must_border_claimed_edge,
                                     struct_x, struct_y, struct_vertex_dir):

        # A structure can only be placed on a vertex if none of the three
        # adjacent vertices are occupied aka the Distance Rule.
        if new_value.position_type == PositionType.VERTEX:

            adjacent_vertex_vals = \
                self.get_adjacent_vertices_for_vertex(x, y, placement_dir)

            adjacent_structures = filter(
                lambda vertex_val: isinstance(vertex_val, Structure),
                adjacent_vertex_vals
            )

            if len(adjacent_structures):
                raise InvalidStructurePlacementException()

        # If the struct_x etc. are provided, they specify a vertex the new
        # edge to place must border e.g. as in initial placement stage.
        if new_value.position_type == PositionType.EDGE and \
                        struct_x is not None:
            allowable_edges = self.get_adjacent_edges(struct_x, struct_y, struct_vertex_dir)
            target_edge = self.get_tile_with_coords(x, y).get_edge(placement_dir)

            if target_edge not in allowable_edges:
                raise InvalidStructurePlacementException()

        # If the player is replacing an existing structure...
        if isinstance(old_value, Structure):

            # The old structure must be owned by the same player.
            if old_value.owning_player != new_value.owning_player:
                raise BoardPositionOccupiedException((x, y), old_value,
                                                     old_value.owning_player)

            # The new value must be an augmenting structure whose base structure
            # matches the existing structure.
            if (not new_value.is_augmenting_structure()) or \
                    (new_value.is_augmenting_structure() and \
                     old_value.name != new_value.augments):
                raise InvalidBaseStructureException(old_value, new_value)

        # If the player is not replacing an existing structure, make sure it’s
        # neighboring a road, unless overridden e.g. as during initial
        # structure placement.
        elif must_border_claimed_edge:
            if placement_dir in EdgeDirection:
                edge_vals = self.get_adjacent_edges_for_edge(x, y, placement_dir)
            elif placement_dir in VertexDirection:
                edge_vals = self.get_adjacent_edges_to_vertex(x, y, placement_dir)

            claimed_edge_structs = filter(
                lambda edge_val: isinstance(edge_val, Structure) and
                                 edge_val.owning_player == new_value.owning_player,
                edge_vals
            )

            if not len(claimed_edge_structs):
                raise InvalidStructurePlacementException()

    def distribute_resources_for_roll(self, roll_value):
        """Distribute resources to the players based on the given roll value.

        Resources are distributed as follows: Whenever a value is rolled that
        matches the chit value of a tile, for all structures on that tile,
        distribute the number of resources dictated by the yield of that
        structure of the type of that tile.

        Args:
            roll_value (int): Dice roll value used to determine which tiles
              should yield resources this turn.

        Returns:
            dict. Primary keys are players and secondary keys are resource
              types. Stored values are the number of a given resource that was
              distributed to the player.
        """

        # Find those tiles whose chit value matches the roll value,
        # and whose yield isn’t blocked by a calamity.
        resource_tiles = filter(
            lambda tile:
                tile.chit_value == roll_value and
                (CalamityTilePlacementEffect.BLOCK_YIELD not in
                    tile.get_calamity_tile_placement_effects()),
            list(self.iter_tiles())
        )

        distributions = Utils.nested_dict()

        # Create a dictionary that stores per-player resource distributions.
        # i.e. distributions => player => resource_type => (int)
        for resource_tile in resource_tiles:

            # Find any structures built on the vertices of the found tiles.
            adjacent_structures = resource_tile.get_adjacent_vertex_structures()

            for structure in adjacent_structures:
                player = structure.owning_player
```

```python
                resource_type = resource_tile.resource_type
                resource_yield = structure.base_yield

                if not distributions[player][resource_type]:
                    distributions[player][resource_type] = 0

                distributions[player][resource_type] += resource_yield

        self.distribute_resources(distributions)

        return distributions

    def distribute_resources(self, distributions):

        # Now distribute resources to players, if the bank has enough.
        for resource_type in ResourceType.get_arable_types():

            def get_per_player_production(player):
                resource_count = distributions[player][resource_type]
                return resource_count if resource_count else 0

            total_count = sum(map(get_per_player_production, distributions))

            try:
                self.bank.withdraw_resources(resource_type, total_count)

                for player in distributions:

                    count = distributions[player][resource_type]

                    if count:
                        player.deposit_resources(resource_type, count)

            except NotEnoughResourcesException:
                # Bank didn’t have enough of the current resource to distribute
                # to all players, so distribute none of this resource.
                pass

        return distributions

    def find_robber(self):
        """Return the robber we can find."""

        for tile in self.iter_tiles():
            for calamity in tile.calamities:
                if isinstance(calamity, Robber):
                    return calamity

        return None

    def get_tile_of_resource_type(self, resource_type):
        """Returns first found file of specified resource type."""

        for tile in self.iter_tiles():
            if tile.resource_type == resource_type:
                return tile

        return None

    def find_tile_with_calamity(self, calamity):

        for tile in self.iter_tiles():
            if calamity in tile.calamities:
                return tile

        return None

    def place_calamity(self, x, y, calamity):

        tile = self.get_tile_with_coords(x, y)
        tile.add_calamity(calamity)
```

```
# -*- coding: utf-8 -*-
import pdb

from engine.src.lib.utils import Utils
from engine.src.board.board import Board
from engine.src.tile.hex_tile import HexTile
from engine.src.vertex import Vertex
from engine.src.edge import Edge
from engine.src.direction.edge_direction import EdgeDirection
from engine.src.direction.vertex_direction import VertexDirection
from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping


class HexBoard(Board):
    """A horizontal hextile board, such as that used in Settlers of Catan.

    Hextiles are referred to using axial coordinates.
        See below for more on axial hex coordinates.
            http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/
            www.redblobgames.com/grids/hexagons

    Attributes:
        radius (int): The number of tiles between the center tile and the edge
          of the board, including the center tile itself. Should be >= 1.

        tiles (dict): A dictionary of tiles, indexed using axial coordinates

        tile_cls (class): Class of the tiles to be generated during board
          initialization.

    Args:
        radius (int): The number of tiles between the center tile and the edge
          of the board, including the center tile itself. Should be >= 1.
    """

    MIN_BOARD_RADIUS = 1

    def __init__(self, radius, tile_cls=HexTile):

        if radius < HexBoard.MIN_BOARD_RADIUS:
            message = ("Specified radius does not meet the minimum board "
                       "tile radius {0}").format(HexBoard.MIN_BOARD_RADIUS)
            raise ValueError(message)

        self.radius = radius

        self.tile_cls = tile_cls

        self.tiles = {}
        self._create_tiles()

    def _create_tiles(self):
        """Generates a dictionary of tiles, indexed by axial coordinates.

        See how coordinates are generated in _add_new_tile_with_coords()

        Returns:
            None.
        """

        for x, y in self.iter_tile_coords():
            self._add_new_tile_with_coords(x, y)

        self._sync_tile_vertices_and_edges()


    def _add_new_tile_with_coords(self, x, y):
        """Add a brand new tile to the board at the given axial coordinates."""

        if x not in self.tiles:
            self.tiles[x] = {}

        tile = self.tile_cls(x, y)
        self.tiles[x][y] = tile

    def _sync_tile_vertices_and_edges(self):
        """Synchronize shared vertices and edges across tiles.

        New tile objects will create their own vertices and edges. When tiles
        share edges and vertices with existing tiles on the board, however,
        we want them to point to the same shared vertex or edge objects,
        instead of each having their own. This method enforces this for the
        given tile.
        """

        for x, y in self.iter_tile_coords():
            tile = self.get_tile_with_coords(x, y)

            for vertex_dir in VertexDirection:
                new_vertex = Vertex()
                self.update_vertex(x, y, vertex_dir, new_vertex)

            for edge_dir in EdgeDirection:
                new_edge = Edge()
                self.update_edge(x, y, edge_dir, new_edge)

    def get_tile_with_coords(self, x, y):
        """Get the tile at the given coordinates, or None if no tile exists."""

        if x in self.tiles and y in self.tiles[x]:
            return self.tiles[x][y]

        return None

    def get_vertex(self, x, y, vertex_dir):
        """Get the vertex defined by the given params."""
        tile = self.get_tile_with_coords(x, y)

        if tile:
            return tile.vertices[vertex_dir]
        else:
            return None

    def valid_tile_coords(self, x, y):
        """Return whether or not these params specify a valid tile."""

        return bool(self.get_tile_with_coords(x, y))

    def valid_vertex(self, x, y, vertex_dir):
        """Return whether or not these params specify a valid vertex."""

        return bool(self.get_vertex(x, y, vertex_dir))

    def get_neighboring_tile(self, tile, edge_direction):
        """Get the tile neighboring the given tile in the given direction.

        Args:
            tile (Tile): The tile for which we’d like to find the neighbor.

            edge_direction (EdgeDirection): Hextiles have 6 edges and thus
              neighbors in 6 different directions. Should be relative to the
              given tile.
        Returns:
            Tile. None if the tile has no valid neighbor in that direction.

        TODO: enforce that direction is actually in EdgeDirection
        """

        x = tile.x + edge_direction[0]
        y = tile.y + edge_direction[1]

        return self.get_tile_with_coords(x, y)

    def get_neighboring_tiles(self, tile):
        """Get all six neighboring tiles for the given hextile.

        Args:
            tile (Tile): The tile whose neighbors we want to return.

        Returns:
            dict. Keys are directions and values are tiles that neighbor the
              given tile in that direction.
        """

        neighboring_tiles = {}

        for direction in EdgeDirection:
            neighbor_tile = self.get_neighboring_tile(tile, direction)

            if neighbor_tile:
                neighboring_tiles[direction] = neighbor_tile

        return neighboring_tiles

    def iter_tiles(self):
        """Iterate over the tiles in this board.

        The order is that described in iter_tile_coords.

        Yields:
            Tile. Each tile of the board.
        """

        for x, y in self.iter_tile_coords():
            yield self.get_tile_with_coords(x, y)

    def iter_perimeter_tiles(self):
        """Iterate over the tiles along the outermost edge of the board."""
        for x, y in HexBoard.iter_tile_ring_coords(self.radius - 1):
            yield self.get_tile_with_coords(x, y)

    def iter_tile_coords(self):
        """Iterate over axial coordinates for each tile in the board.

        This is a generator function that will yield the coordinates to the
        caller each time after they are computed.

        We can consider a hextile board a series of concentric rings where the
        radius counts the number of concentric rings that compose the board.
        When generating coordinates, we traverse each such ring one at a time,
        using the pattern specified in iter_tile_ring_coords().

        Yields:
            tuple. The axial (x, y) coordinates of each tile on the board.
        """

        for ring_index in range(self.radius):
            for x, y in HexBoard.iter_tile_ring_coords(ring_index):
                yield x, y

    @staticmethod
    def iter_tile_ring_coords(ring_index):
        """Iterate clockwise over coordinates of the board’s perimeter tiles.

        We can consider a hextile board a series of concentric rings where the
        radius counts the number of concentric rings that compose the board.
        Thus, ring_index 0 corresponds to the center tile and ring_index =
        self.radius - 1 corresponds to perimeter tiles.

        Here we generate the coordinates for all tiles of a single ring,
        designated by ring_index, traversing the ring one tile at a time,
        starting from the westernmost tile and continuing around the ring in a
        clockwise fashion.

        Args:
            ring_index (int): Defines which tile ring to iterate over.
              Should be a value between 0 and self.radius - 1.

        Yields:
            tuple. The axial (x, y) coordinates of each tile in the given ring.
        """

        # We start yielding coordinates from the westernmost tile.
        x = -1 * ring_index
        y = 0

        if x == 0 and y == 0:
            yield x, y

        # First we scale the northwest side of the ring.
        # This is equivalent to moving along the y-axis of the board.
        while y != ring_index:
            yield x, y
            y += 1

        # Then we scale the northern side of the ring.
        # This is equivalent to moving along the x-axis of the board.
        while x != 0:
            yield x, y
            x += 1

        # Then we scale the northeast side of the ring.
        # This is equivalent to moving along the z-axis of the board.
        while x != ring_index or y != 0:
            yield x, y
            x += 1
            y -= 1

        # Then we scale the southeast side of the ring.
        while y != -ring_index:
            yield x, y
            y -= 1

        # Then the south side of the ring.
        while x != 0:
            yield x, y
            x -= 1

        # And finally the south west side of the ring.
        while x != -ring_index:
            yield x, y
            x -= 1
            y += 1

    def update_edge(self, x, y, edge_dir, edge_val):
        """Update the specified edge.

        Also updates equivalent edge for neighboring tile.

        Args:
            x (int): Axial x-coordinate of the tile, one of whose vertices
              we will update.

            y (int): Axial y-coordinate of the tile, one of whose vertices
              we will update.

            edge_dir (EdgeDirection): Direction of edge to update relevant to
              tile given by x, y coordinates.

            edge_val (Structure): Value to replace old edge values.

        Returns:
            None
        """
        tile = self.get_tile_with_coords(x, y)
        vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)

        neighbor_tile = self.get_neighboring_tile(tile, edge_dir)

        tile.add_edge(vertex_dirs[0], vertex_dirs[1], edge_val)

        # Perimeter tiles will not have neighbors along certain edges.
        if neighbor_tile:
            nv_dir_1 = HexTile.get_equivalent_vertex_dir(vertex_dirs[0], edge_dir)
            nv_dir_2 = HexTile.get_equivalent_vertex_dir(vertex_dirs[1], edge_dir)
            neighbor_tile.add_edge(nv_dir_1, nv_dir_2, edge_val)

    def update_vertex(self, x, y, vertex_dir, vertex_val):
        """Update the value at the specified vertex location.

        Also updates vertex for neighboring tiles.

        Args:
            x (int): Axial x-coordinate of the tile, one of whose vertices
              we will update.

            y (int): Axial y-coordinate of the tile, one of whose vertices
              we will update.

            vertex_dir (VertexDirection): Vertex direction, relative to the
              tile specified by the x and y coordinates, of the vertex to
              update.

            vertex_val (Structure): Value to replace old vertex values.

        Returns:
            None.
        """

        tile = self.get_tile_with_coords(x, y)
        old_vertex_val = self.get_vertex(x, y, vertex_dir)

        tile.vertices[vertex_dir] = vertex_val

        # Get the two edges of the found tile that have as an endpoint
        # a vertex of the given vertex direction.
        vertex_adj_edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(
            vertex_dir)

        for vertex_adj_edge_dir in vertex_adj_edge_dirs:
            neighbor_tile = self.get_neighboring_tile(tile, vertex_adj_edge_dir)

            # Edge tiles may not have neighboring tiles in the given direction.
            if neighbor_tile:
                neighbor_vertex_dir = HexTile.get_equivalent_vertex_dir(
                    vertex_dir, vertex_adj_edge_dir)

                neighbor_tile.update_vertex(neighbor_vertex_dir, vertex_val)

    def get_adjacent_tiles_to_vertex(self, x, y, vertex_dir):
        """Get the three tiles that converge at the specified vertex.

        Args:
            x (int): Axial x-coordinate of the tile, one of whose vertices
              we will update.

            y (int): Axial y-coordinate of the tile, one of whose vertices
              we will update.

            vertex_dir (VertexDirection): Vertex direction, relative to the
              tile specified by the x and y coordinates, of the vertex to
              find the adjacent tiles of.

        Returns:
            list of Tiles. The tiles that converge at the specified vertex.
        """

        tile = self.get_tile_with_coords(x, y)

        adjacent_tiles = map(
            lambda edge_dir: self.get_neighboring_tile(tile, edge_dir),
            EdgeVertexMapping.get_edge_dirs_for_vertex_dir(vertex_dir)
        )

        adjacent_tiles.append(tile)

        return adjacent_tiles

    def get_adjacent_edges(self, x, y, vert_or_edge_dir, return_values=True):
        if vert_or_edge_dir in EdgeDirection:
            if return_values:
                return self.get_adjacent_edges_for_edge(x, y, vert_or_edge_dir)
            else:
                return self._get_adjacent_edges_for_edge(x, y, vert_or_edge_dir)

        elif vert_or_edge_dir in VertexDirection:
            if return_values:
                return self.get_adjacent_edges_to_vertex(x, y, vert_or_edge_dir)
            else:
                return self._get_adjacent_edges_to_vertex(x, y, vert_or_edge_dir)

    def _get_adjacent_edges_to_vertex(self, x, y, vertex_dir):

        tile = self.get_tile_with_coords(x, y)

        edge_vals = []

        # Get the directions of edges that both have vertex_dir as an endpoint.
        edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(vertex_dir)

        edge_vals.append( (x, y, edge_dirs[0]) )
        edge_vals.append( (x, y, edge_dirs[1]) )

        # The last edge value won’t be available via the current tile’s edges,
        # but must be found on its neighbor.
        neighbor_x = tile.x + edge_dirs[0][0]
        neighbor_y = tile.y + edge_dirs[0][1]
        neighboring_tile = self.get_neighboring_tile(tile, edge_dirs[0])
        opp_vert_dir = HexTile.get_equivalent_vertex_dir(vertex_dir, edge_dirs[0])

        neighbor_edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(opp_vert_dir)
        neighbor_edge_dir = next(d for d in neighbor_edge_dirs if d not in \
            map(lambda edge_val: edge_val[2].get_opposite_direction(), edge_vals))

        edge_vals.append( (neighbor_x, neighbor_y, neighbor_edge_dir) )

        return edge_vals

    def get_adjacent_edges_to_vertex(self, x, y, vertex_dir):

        edge_tuples = self._get_adjacent_edges_to_vertex(x, y, vertex_dir)
        edge_vals = []

        msg = "Edges adjacent to ({}, {}) {}:\n".format(x, y, vertex_dir)

        for x, y, edge_dir in edge_tuples:
            tile = self.get_tile_with_coords(x, y)
            edge_val = tile.get_edge(edge_dir)

            edge_vals.append(edge_val)
            msg += ’\t\t ({}, {}) {}\n’.format(x, y, edge_dir)

        return edge_vals

    def _get_adjacent_edges_for_edge(self, x, y, edge_dir):

        vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)

        edge_tuples = []
        edge_tuples.extend(self._get_adjacent_edges_to_vertex(x, y, vertex_dirs[0]) + \
                           self._get_adjacent_edges_to_vertex(x, y, vertex_dirs[1]))

        edge_tuples = filter(
            lambda edge_tuple: edge_tuple[2] != edge_dir,
            edge_tuples
        )

        return edge_tuples

    def get_adjacent_edges_for_edge(self, x, y, edge_dir):

        edge_tuples = self._get_adjacent_edges_for_edge(x, y, edge_dir)
        edge_vals = []

        for ex, ey, e_dir in edge_tuples:
            tile = self.get_tile_with_coords(ex, ey)

            if tile:
                edge_vals.append(tile.get_edge(e_dir))

        return edge_vals

    def _get_adjacent_vertices_for_vertex(self, x, y, vertex_dir):

        vertex_tuples = []

        tile = self.get_tile_with_coords(x, y)

        vertex_dirs = VertexDirection.get_neighboring_vertex_dirs(vertex_dir)

        # Two of the closest vertices will lie on this tile
        for adjacent_vertex_dir in vertex_dirs:
            vertex_tuple = (x, y, adjacent_vertex_dir)
            vertex_tuples.append(vertex_tuple)

        # The last vertex value won’t be available via the current tile’s
        # vertices, but must be found on its neighbor.

        edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(vertex_dir)

        # Pick one edge, arbitrarily, to find the neighbor tile relative to that edge.
        neighbor_edge_dir = edge_dirs[0]
        neighboring_tile = self.get_neighboring_tile(tile, neighbor_edge_dir)
        neighbor_x = tile.x + neighbor_edge_dir[0]
        neighbor_y = tile.y + neighbor_edge_dir[1]

        # Find the neighbor equivalent of vertex_dir
        opp_vert_dir = HexTile.get_equivalent_vertex_dir(vertex_dir, neighbor_edge_dir)

        # Vertex and edge direction should be relative to same tile
        def vertex_already_found(v_dir, neighbor_edge):
            neighbor_equivalent_v_dir = \
                HexTile.get_equivalent_vertex_dir(v_dir, neighbor_edge_dir.get_opposite_direction())
            return neighbor_equivalent_v_dir not in map(lambda v_tup: v_tup[2], vertex_tuples)

        # Find the vertices adjacent to neighbors equivalent of vertex_dir.
        # One will duplicate a vertex we already have, one will be new.
        # Filter out the duplicate.
        last_vertex_dir = filter(
            lambda v_dir: not vertex_already_found(v_dir, neighbor_edge_dir.get_opposite_direction()),
            VertexDirection.get_neighboring_vertex_dirs(opp_vert_dir)
        )

        if len(last_vertex_dir):
            last_vertex_dir = last_vertex_dir[0]
            vertex_tuples.append( (neighbor_x, neighbor_y, last_vertex_dir) )

        return vertex_tuples

    def get_adjacent_vertices_for_vertex(self, x, y, vertex_dir):

        vertex_tuples = self._get_adjacent_vertices_for_vertex(x, y, vertex_dir)
        vertex_vals = []

        for vx, vy, v_dir in vertex_tuples:
            tile = self.get_tile_with_coords(vx, vy)

            if tile:
                vertex_vals.append(tile.get_vertex(v_dir))

        return vertex_vals
```

```
# -*- coding: utf-8 -*-
from abc import ABCMeta, abstractmethod, abstractproperty
from enum import Enum


class Calamity(object):
    """
    TODO: Consider breaking Calamity subclasses based on their latent effect,
          i.e. when not rolled, but on the board. So robbers block tile yield.
          Other calamities might block structure construction.
    """
    __metaclass__ = ABCMeta

    DEFAULT_ROLL_VALUES = [7]

    @abstractproperty
    def roll_value(self):
        """The dice roll value that should trigger this calamity’s effect."""
        pass

    @abstractmethod
    def trigger_effect(self, game, player):
        """Activates this calamity’s effect.

        Args:
            game (Game): The game this calamity will affect.

            player (Player): Player who rolled the triggering roll.
        """
        pass


class CalamityTilePlacementEffect(Enum):
    BLOCK_YIELD = 1
```

```
# -*- coding: utf-8 -*-
from engine.src.calamity.calamity import Calamity
from engine.src.calamity.calamity import CalamityTilePlacementEffect


class Robber(Calamity):

    MIN_ROBBER_ACTIVATING_RESOURCE_COUNT_THRESHOLD = 8

    def __init__(self):
        # TODO: Not sure if this is the best way to represent these effects.
        self.tile_placement_effect = CalamityTilePlacementEffect.BLOCK_YIELD

    def roll_value(self):
        # TODO: Move to config?
        return 7

    def trigger_effect(self, game, player):
        """Halve players resources, move the robber, draw a resource card.

        Triggering the robber effect elicits the following behavior:
            (1) All players who have more than some threshold of resource cards
                must discard half of their resource hand, floored.
            (2) See self.outside_trigger_effect().

        Args:
            See Calamity.
        """

        threshold = Robber.MIN_ROBBER_ACTIVATING_RESOURCE_COUNT_THRESHOLD

        # Have players discard half their hand if they have too many cards.
        for game_player in game.players:

            resource_count = game_player.count_resources()

            if resource_count > threshold:
                cards_to_discard = int(resource_count / 2)
                resources = game_player.get_resource_list()

                resource_indices = game.input_manager.prompt_discard_resources(
                    game, player, resources, cards_to_discard)

                for index in resource_indices:
                    game_player.withdraw_resources(resources[index], 1)

        self.outside_trigger_effect(game, player)

    def outside_trigger_effect(self, game, player):
        """When the robber is activated not by a dice roll, call this method.

        Execute the following behavior:
            (1) The robber should be moved to a different tile.
            (2) A resource card must be drawn from one of the players with
                structures built adjacent to the tile.
        """

        robber_successfully_moved = False
        previous_tile = game.board.find_tile_with_calamity(self)
        previous_tile.remove_calamity(self)

        tile = None

        prompt = ’Select a tile to move the robber to. Current location: {0}’\
            .format(previous_tile)

        game.input_manager.input_default(prompt, None, False)

        while not robber_successfully_moved:
            x, y = game.input_manager.prompt_tile_coordinates(game)

            # Move robber to new tile.
            tile = game.board.get_tile_with_coords(x, y)

            if tile != previous_tile:
                tile.add_calamity(self)
                robber_successfully_moved = True

        # Draw card from player that has a structure built adjacent to the tile.
        # The player can not draw from herself or from a player with no cards.
        eligible_players = filter(
            lambda owning_player:
                owning_player != player and
                owning_player.count_resources() != 0,
            map(lambda structure: structure.owning_player,
                tile.get_adjacent_vertex_structures())
        )

        if eligible_players:

            # Chose a player to randomly select a resource from.
            chosen_player = game.input_manager.prompt_select_player(
                game, eligible_players)

            resource_type = chosen_player.withdraw_random_resource()
            player.deposit_resources(resource_type, 1)

            # Announce received resource.
            msg = ’You received 1 {0} from {1}.’.format(
                resource_type, chosen_player.name)
            game.input_manager.input_default(msg, None, False)

        else:
            # Announce no eligible players to draw from.
            msg = ’No qualifying players to draw from.’
            game.input_manager.input_default(msg, None, False)
```

```
# -*- coding: utf-8 -*-
from engine.src.config.config import Config
from engine.src.lib.utils import Utils


class DevelopmentCard(object):
    """
    Attributes:
        From Config:
            count (int)
            name (str)
            description (str)
            draw_card (func)
            play_card (func)
            cost (int)

        played (bool)
        is_playable (bool)
    """

    def __init__(self, **kwargs):

        # Initialize default values.
        Config.init_from_config(self, 'game.card.development.default')

        # Overwrite default values with custom values.
        Utils.init_from_dict(self, kwargs)

        self.played = False
        self.is_playable = True

    def __str__(self):
        return self.name

    def draw_card(self, game, player):
        """Draw this card and activate any effect incurred by holding it.

         This method should be called only once when purchased by a player.

        Args:
            game (Game): The game this card may possibly affect.

            player(Player): The player that bought this development card.

        Returns:
            None. Should call functions on game and player.
        """
        pass

    def play_card(self, game, player):
        """Draw this card and activate any relevant effect.

         This method should be called only once when played by a player.

        Args:
            game (Game): The game this card may possibly affect.

            player(Player): The player that played this development card.

        Returns:
            None. Should call functions on game and player.
        """

        self.played = True
```

```python
def draw_card(self, game, player):
    pass


def play_card(self, game, player):
    """Move the robber and draw a card from another adjacent player."""

    game.input_manager.announce_development_card_played(player, self)

    robber = game.board.find_robber()

    robber.outside_trigger_effect(game, player)

    player.knights += 1

    self.played = True
```

```
def draw_card(self, game, player):
    pass


def play_card(self, game, player):
    """Allow player to take all carried cards of selected resource type."""

    game.input_manager.announce_development_card_played(player, self)
    resource_type = game.input_manager.prompt_select_resource_type()

    for game_player in game.players:
        if player != game_player:
            count = player.resources[resource_type]

            game_player.transfer_resources(player, resource_type, count)

            msg = '{0} received {1} {2} from {3}'.format(
                player.name, count, resource_type, game_player.name)

            game.input_manager.input_default(msg, None, False)

    # Announce finished collecting resources.
    msg = 'Done monopolizing resources.'
    game.input_manager.input_default(msg, None, False)

    self.played = True
```

```
def draw_card(self, game, player):
    pass


def play_card(self, game, player):
    """Allow player to take all carried cards of selected resource type."""

    game.input_manager.announce_development_card_played(player, self)

    for _ in range(2):
        x, y, edge_dir = game.input_manager.prompt_edge_placement(game)
        game.board.place_edge_structure(x, y, edge_dir,
                                        player.get_structure(’road’))

    self.played = True
```

```
def draw_card(self, game, player):
    player.hidden_points += 1


def play_card(self, game, player):
    # We could convert the player’s hidden points to public points,
    # but keeping the points hidden makes it easier to recompute
    # a player’s overall point total from scratch.
    pass
```

```python
def draw_card(self, game, player):
    pass


def play_card(self, game, player):
    """Allow player to take 2 cards of their chosen resource type."""

    game.input_manager.announce_development_card_played(player, self)
    resource_type = game.input_manager.prompt_select_resource_type()

    game.board.bank.transfer_resources(player, resource_type, 2)

    self.played = True
```

```python
from types import *
from engine.src.lib.utils import Utils
from engine.src.config.game_config import game_config
from engine.src.config.type_config import type_config
from engine.src.config.type_mapping import type_mapping
from engine.src.exceptions import *
import pdb


class Config(object):

    is_coerced = False

    @classmethod
    def init_from_config(cls, obj, config_path):
        property_dict = Config.get(config_path)
        dct = { Utils.convert_format(k): v for (k, v) in property_dict.iteritems()}
        Utils.init_from_dict(obj, dct)

    @classmethod
    def pluck(cls, config_path, prop):
        target_dict = Config.get(config_path)
        return Utils.pluck(target_dict, prop, True)

    @classmethod
    def set(cls, value, dot_notation_str, dct=None):

        if dct is None:
            dct = Config.config

        keys = dot_notation_str.split(’.’)

        def set_recursive(dct, keys):
            if not keys:
                return dct

            key = keys.pop(0)
            val = None

            if key in dct:
                val = dct.get(key)
            else:
                raise NoConfigValueDefinedException(dot_notation_str)

            # If we still have keys left, the property we want to set is nested
            # somewhere inside the value we fetched.
            if keys:
                if val:
                    return set_recursive(val, keys)
                else:
                    raise NoConfigValueDefinedException(dot_notation_str)
            # If we have no keys left, we’ve found the target value.
            else:
                dct[key] = value

        set_recursive(dct, keys)


    @classmethod
    def get(cls, dot_notation_str, dct=None, remove_default=True):
        """Get a value from the main config dict given a dot notation string.

        E.g. if caller wants config[’game’][’points_to_win’], they can pass in
        as their dot_notation_str ’game.points_to_win’.

        See coerce() for effect of coerce_type flag.
        """

        if not Config.is_coerced:
            Config.coerce_all()

        if dct is None:
            dct = Config.config

        if not dot_notation_str:
            return dct

        keys = dot_notation_str.split(’.’)

        def get_recursive(dct, keys):
            key = keys.pop(0)
            val = None

            # Get the value of the key if it’s in the dict.
            if key in dct:
                val = dct.get(key)
            elif key.replace(’_’, ’-’) in dct:
                val = dct.get(key.replace(’_’, ’-’))
            else:
                # print "loc: {}\ndct: {}\nkey: {}".format(dot_notation_str, dct, key)

                # print Config.config
                raise NoConfigValueDefinedException(dot_notation_str)

            # If we still have keys left, the property we want is nested
            # somewhere inside the value we fetched.
            if keys:
                if val:
                    return get_recursive(val, keys)
                else:
                    raise NoConfigValueDefinedException(dot_notation_str)
            # If we have no keys left, we’ve found the target value.
            else:
                return val

        value = get_recursive(dct, keys)

        if remove_default:
            # Remove default value from dictionary type return value.
            if type(value) is dict:
                value = {k: value[k] for k in value.keys() if k != ’default’}

        return value

    @classmethod
    def init(cls):
        Config.convert_keys()
        Config.coerce_all()

    @classmethod
    def convert_keys(cls):

        def convert(dct):
            for k, v in dct.iteritems():

                if type(k) is StringType:
                    dct.pop(k)
                    dct[Utils.convert_format(k)] = v

                if type(v) is dict:
                    convert(v)

        convert(Config.config)

    @classmethod
    def coerce_all(cls):
        Config.is_coerced = True
        Config.coerce_recursive(’’)

    @classmethod
    def coerce_recursive(cls, path_so_far):
        curr_value = Config.get(path_so_far, Config.config, False)

        try:
            target_type = Config.get(
                Config.get_default_path(path_so_far), Config.type_config, False)
        except NoConfigValueDefinedException:
            return

        is_struct = False

        if type(curr_value) is dict:
            is_struct = len(filter(
                lambda key: type(key) == StringType,
                target_type.keys()
            )) != 0

        if is_struct:
            for k, v in curr_value.iteritems():
                path = k if not path_so_far else ’.’.join([path_so_far, k])
                Config.coerce_recursive(path)
        else:
            # print "Beginning coercion, path: {}".format(path_so_far)
            # print "Current type: {}".format(type(curr_value))
            # print "Target type: {}".format(target_type)
            Config.set(
                Config.coerce(curr_value, type(curr_value), target_type),
                path_so_far
            )

    @classmethod
    def coerce(cls, value, from_type, to_type):

        if from_type == to_type:
            return value

        if from_type is dict:
            result = {}

            target_k_type = to_type.keys()[0]
            target_v_type = to_type.values()[0]

            for k, v in value.iteritems():
                coerced_k_value = Config.coerce(k, type(k), target_k_type)
                coerced_v_value = Config.coerce(v, type(v), target_v_type)

                result[coerced_k_value] = coerced_v_value

            return result
        else:
            coercion_func = type_mapping[from_type][to_type]
            return coercion_func(value)

    @classmethod
    def get_default_path(cls, dot_notation_str):
        # e.g. structure.player_built.road.cost =>
        #      structure.player_built.default.cost
        """
        If last prop is not a dict, replace second to last with default
        If last prop is a dict, e.g. structure.player_built.road
            if dict is a struct, replace last with default
            if dict isn’t a struct, replace second to last with default
        """

        value = None
        path = None

        repl_index = -1

        while True:
            keys = dot_notation_str.split(’.’)

            try:
                keys[repl_index] = ’default’
                path = ’.’.join(keys)
                value = Config.get(path, Config.type_config, False)
                break
            except NoConfigValueDefinedException:
                repl_index -= 1
            except IndexError:
                # No defaults; return as is.
                path = dot_notation_str
                break

        return path

    # The dictionary accessed by Config.get()
    config = {}

    # A dictionary telling us what object types we should expect
    # for values in config.
    type_config = type_config

    type_mapping = type_mapping
```

```
from engine.src.resource_type import ResourceType
from engine.src.lib.utils import Utils

def get_import_value(dot_notation_str, var_name, prefix='engine.src.config.'):
    mod = __import__(prefix + dot_notation_str, globals(), locals(), [var_name], -1)
    value = getattr(mod, var_name)
    return value

game_config = {
    # Game
    'game' : {
        'points_to_win': 10,
        'player_count': 3,

        'board' : {
            'tile_count': 19,
            'radius': 3,
        },
        # Cards
        'card' : {
            # Development Cards
            'development': {
                'default': {
                    'count': 0,
                    'name': 'Development Card',
                    'description': 'Development card default description.',
                    'draw_card': None,
                    'play_card': None,
                    'cost': {
                        'wool': 1,
                        'grain': 1,
                        'ore': 1
                    },
                },
                # Non-Progress Cards
                'knight': {
                    'count': 14,
                    'name': 'Knight Card',
                    'description': ('Move the robber to a new tile. Steal 1 '
                                    'resource from the owner of a structure '
                                    'adjacent to the new tile.'),
                    'draw_card': get_import_value('card.development.knight', 'draw_card'),
                    'play_card': get_import_value('card.development.knight', 'play_card'),
                },
                'victory_point': {
                    'count': 5,
                    'name': 'Victory Point Card',
                    'description': ('Gives you one victory point. Must remain '
                                    'hidden until used to win the game.'),
                    'draw_card':
                        get_import_value('card.development.victory_point', 'draw_card'),
                    'play_card':
                        get_import_value('card.development.victory_point', 'play_card'),
                },
                # Progress Cards
                'monopoly': {
                    'count': 2,
                    'name': 'Monopoly Card',
                    'description': ('If you play this card, you must name 1 type '
                                    'of resource. All the other players must give '
                                    'you all of the Resource Cards of this type '
                                    'that they have in their hands. If an opponent '
                                    'does not have a Resource Card of the '
                                    'specified type, he does not have to give you '
                                    'anything.'),
                    'draw_card': get_import_value('card.development.monopoly', 'draw_card'),
                    'play_card': get_import_value('card.development.monopoly', 'play_card'),
                },
                'road_building': {
                    'count': 2,
                    'name': 'Road Building Card',
                    'description': ('If you play this card, you may immediately '
                                    'place 2 free roads on the board (according to '
                                    'normal building rules)'),
                    'draw_card':
                        get_import_value('card.development.road_building', 'draw_card'),
                    'play_card':
                        get_import_value('card.development.road_building', 'play_card'),
                },
                'year_of_plenty': {
                    'count': 2,
                    'name': 'Year of Plenty Card',
                    'description': ('If you play this card you may immediately '
                                    'take any 2 Resource Cards from the supply '
                                    'stacks. You may use these cards to build in '
                                    'the same turn.'),
                    'draw_card':
                        get_import_value('card.development.year_of_plenty', 'draw_card'),
                    'play_card':
                        get_import_value('card.development.year_of_plenty', 'play_card'),
                }
            }
        },
        # Structures
        'structure': {
            'player_built': {
                'default': {
                    'name': None,
                    'cost': {
                        'lumber': 0,
                        'brick': 0,
                        'wool': 0,
                        'grain': 0,
                        'ore': 0
                    },
                    'count': 0,
                    'point_value': 0,
                    'base_yield': 1,
                    # TODO: Rename vars to reflect that they should be structure names?
                    'extends': None,
                    'upgrades': None,
                    'position_type': 'vertex'
                },
                # Edge Structures
                'road': {
                    'name': 'Road',
                    'cost': {
                        'lumber': 1,
                        'brick': 1,
                    },
                    'count': 15,
                    ’point_value’: 0,
                    ’base_yield’: 0,
                    ’extends’: None,
                    ’upgrades’: None,
                    ’position_type’: ’edge’
                },
                # Vertex Structures
                ’settlement’: {
                    ’name’: ’Settlement’,
                    ’cost’: {
                        ’lumber’: 1,
                        ’brick’: 1,
                        ’wool’: 1,
                        ’grain’: 1
                    },
                    ’count’: 5,
                    ’point_value’: 1,
                    ’base_yield’: 1,
                    ’extends’: None,
                    ’upgrades’: None,
                    ’position_type’: ’vertex’
                },
                ’city’: {
                    ’name’: ’City’,
                    ’cost’: {
                        ’grain’: 2,
                        ’ore’: 3,
                    },
                    ’count’: 5,
                    ’point_value’: 2,
                    ’base_yield’: 2,
                    ’extends’: None,
                    ’upgrades’: ’Settlement’,
                    ’position_type’: ’vertex’
                },
                # For Demo
                ’castle’: {
                    ’name’: ’Castle’,
                    ’cost’: {
                        ’ore’: 5
                    },
                    ’count’: 2,
                    ’point_value’: 3,
                    ’base_yield’: 3,
                    ’extends’: None,
                    ’upgrades’: ’City’,
                    ’position_type’: ’vertex’
                }
            }
        }
    }
}
```

```
from engine.src.resource_type import ResourceType
from engine.src.position_type import PositionType
from types import *

type_config = {
    ’game’: {
        ’points_to_win’: IntType,
        ’player_count’: IntType,

        ’board’ : {
            ’tile_count’: IntType,
            ’radius’: IntType,
        },
        ’structure’: {
            ’player_built’: {
                ’default’: {
                    ’cost’: {ResourceType: IntType},
                    ’position_type’: PositionType
                }
            }
        },
        ’card’: {
            ’development’: {
                ’default’: {
                    ’cost’: {ResourceType: IntType},
                    ’draw_card’: FunctionType,
                    ’play_card’: FunctionType
                }
            }
        }
    }
}
```

```python
import engine.src.lib.utils as utils
from engine.src.resource_type import ResourceType
from engine.src.position_type import PositionType
from types import *


type_mapping = { # from_type => to_type => conversion function
    StringType: {
        ResourceType: lambda st: ResourceType.find_by_value(st),
        PositionType: lambda st: PositionType.find_by_value(st)
    },
    NoneType: {
        FunctionType: lambda _: utils.noop,
        MethodType: lambda _: utils.Utils.noop
    }
}
```

```python
# -*- coding: utf-8 -*-
import random


class Dice(object):
    """ Represents a set of game dice.

    Args:
        dice_count (int): Number of dice in the game.

        range (list): List of possible dice values.
    """

    def __init__(self, dice_count=2, values=range(1, 7)):
        self.dice_count = dice_count
        self.values = values

    def roll(self):
        """ Rolls dice.

        Returns:
            int. Sum of dice face values after a random throw.
        """

        return sum(random.choice(self.values) for _ in range(self.dice_count))
```

```
__all__ = ['edge_direction', 'vertex_direction']
```

```python
# -*- coding: utf-8 -*-
from enum import Enum


class Direction(Enum):
    """An abstract class that defines basic functions needed by direction enums.

    TODO: Enforce that this class is an abstract class by having
          its metaclass be ABCMeta. This seems to create some issues since
          Enum is not a regular class and comes from a backport.
    """

    def __str__(self):
        return ’{0}: {1}’.format(self.name, self.value)

    def __getitem__(self, index):
        return self.value[index]

    def __len__(self):
        return len(self.value)

    def __iter__(self):
        return iter(self.value)

    def __eq__(self, other):

        if not other or not hasattr(other, ’__len__’):
            return False

        if len(other) != len(self):
            return False

        for index, value in enumerate(self):
            if not value == other[index]:
                return False

        return True

    @classmethod
    def find_by_value(cls, value):
        for direction in cls:
            if value == direction:
                return direction

```

```python
# -*- coding: utf-8 -*-
from engine.src.direction.direction import Direction


class EdgeDirection(Direction):
    """The 6 directions of a hexagon’s edges with axial coordinates.

    Each edge direction is a direction we can follow from the center of a
    hextile to a point on one of its edges.

    Since each edge in a tile borders another tile, each edge direction
    also corresponds to a unit vector that we can follow from a given
    point in a hex axial coordinate system to get to another tile.

    See more on axial coordinates here:
        http://www.redblobgames.com/grids/hexagons/#coordinates
    """

    NORTH_WEST = (-1, 1, 0)
    NORTH_EAST = (0, 1, -1)
    WEST = (-1, 0, 1)
    EAST = (1, 0, -1)
    SOUTH_WEST = (0, -1, 1)
    SOUTH_EAST = (1, -1, 0)

    def get_opposite_direction(self):
        """Get the direction of the opposite edge."""

        coordinates = self.value

        x = -coordinates[0]
        y = -coordinates[1]
        z = -(x + y)

        return EdgeDirection.find_by_value((x, y, z))
```

```python
# -*- coding: utf-8 -*-
from engine.src.direction.edge_direction import EdgeDirection
from engine.src.direction.vertex_direction import VertexDirection


class EdgeVertexMapping(object):

    vertex_edge_mapping = {
        VertexDirection.TOP:
            (EdgeDirection.NORTH_WEST, EdgeDirection.NORTH_EAST),
        VertexDirection.TOP_RIGHT:
            (EdgeDirection.NORTH_EAST, EdgeDirection.EAST),
        VertexDirection.BOTTOM_RIGHT:
            (EdgeDirection.EAST, EdgeDirection.SOUTH_EAST),
        VertexDirection.BOTTOM:
            (EdgeDirection.SOUTH_EAST, EdgeDirection.SOUTH_WEST),
        VertexDirection.BOTTOM_LEFT:
            (EdgeDirection.SOUTH_WEST, EdgeDirection.WEST),
        VertexDirection.TOP_LEFT:
            (EdgeDirection.WEST, EdgeDirection.NORTH_WEST)
    }

    edge_vertex_mapping = {
        EdgeDirection.NORTH_WEST:
            (VertexDirection.TOP_LEFT, VertexDirection.TOP),
        EdgeDirection.NORTH_EAST:
            (VertexDirection.TOP, VertexDirection.TOP_RIGHT),
        EdgeDirection.EAST:
            (VertexDirection.TOP_RIGHT, VertexDirection.BOTTOM_RIGHT),
        EdgeDirection.SOUTH_EAST:
            (VertexDirection.BOTTOM_RIGHT, VertexDirection.BOTTOM),
        EdgeDirection.SOUTH_WEST:
            (VertexDirection.BOTTOM, VertexDirection.BOTTOM_LEFT),
        EdgeDirection.WEST:
            (VertexDirection.BOTTOM_LEFT, VertexDirection.TOP_LEFT)
    }

    @classmethod
    def get_edge_dirs_for_vertex_dir(cls, vertex_dir):
        """Returns directions of edges that share this vertex direction.

        E.g. VertexDirection.TOP is the direction of the vertex that is an
        endpoint of both EdgeDirection.NORTH_WEST and EdgeDirection.NORTH_EAST.

        Returns:
            tuple. A tuple of two directions, each of which has this vertex as
              an endpoint.
        """

        return EdgeVertexMapping.vertex_edge_mapping[vertex_dir]

    @classmethod
    def get_vertex_dirs_for_edge_dir(cls, edge_dir):
        """Get the vertex directions of endpoints of the given edge.

        Returns:
            tuple. A tuple of 2 tuples, each of which is a value in
              VertexDirection that represents the endpoints of the given edge.
        """

        return EdgeVertexMapping.edge_vertex_mapping[edge_dir]
```

```python
# -*- coding: utf-8 -*-
from engine.src.direction.direction import Direction


class VertexDirection(Direction):
    """The 6 directions of a hexagon’s vertices using cubic coordinates.

    Each vertex direction is a direction we can follow from the center of a
    tile to one of its vertexes.

    If we consider the hexagon a cube, the values correspond to the cubic
    (x, y, z) coordinates of the various directions.

    See more on cubic coordinates here:
        http://www.redblobgames.com/grids/hexagons/#coordinates
    """

    TOP = (1, 1, 0)
    TOP_RIGHT = (1, 0, 0)
    BOTTOM_RIGHT = (1, 0, 1)
    BOTTOM = (0, 0, 1)
    BOTTOM_LEFT = (0, 1, 1)
    TOP_LEFT = (0, 1, 0)

    def get_opposite_direction(self):
        """Get the direction of the vertex opposite one of this direction."""

        coordinates = self.value

        def toggle(val):
            """Toggle val between 0 and 1."""
            return int(not bool(val))

        x = toggle(coordinates[0])
        y = toggle(coordinates[1])
        z = toggle(coordinates[2])

        return VertexDirection.find_by_value((x, y, z))

    @classmethod
    def get_neighboring_vertex_dirs(cls, vertex_dir):

        mapping = {
            VertexDirection.TOP:
                (VertexDirection.TOP_LEFT, VertexDirection.TOP_RIGHT),
            VertexDirection.TOP_RIGHT:
                (VertexDirection.TOP, VertexDirection.BOTTOM_RIGHT),
            VertexDirection.BOTTOM_RIGHT:
                (VertexDirection.TOP_RIGHT, VertexDirection.BOTTOM),
            VertexDirection.BOTTOM:
                (VertexDirection.BOTTOM_RIGHT, VertexDirection.BOTTOM_LEFT),
            VertexDirection.BOTTOM_LEFT:
                (VertexDirection.BOTTOM, VertexDirection.TOP_LEFT),
            VertexDirection.TOP_LEFT:
                (VertexDirection.BOTTOM_LEFT, VertexDirection.TOP),
        }

        return mapping[vertex_dir]

    @classmethod
    def pairs(cls):
        """Returns vertex pairs, each of which constitute an edge of a hex."""

        return (
            (cls.TOP, cls.TOP_RIGHT),
            (cls.TOP_RIGHT, cls.BOTTOM_RIGHT),
            (cls.BOTTOM_RIGHT, cls.BOTTOM),
            (cls.BOTTOM, cls.BOTTOM_LEFT),
            (cls.BOTTOM_LEFT, cls.TOP_LEFT),
            (cls.TOP_LEFT, cls.TOP)
        )
```

```python
# -*- coding: utf-8 -*-


class Edge(object):
    pass
```

```
from engine.src.lib.utils import Utils


class UserMessageException(Exception):
    """
    A custom exception class that prints self.msg when cast to a string.
    """
    def __init__(self, msg):
        self.msg = msg

    def __str__(self):
        return self.msg


class NotEnoughResourcesException(UserMessageException):
    """Raise when a trader lacks enough resources cards for a transaction.

    E.g. when a player doesn’t have enough resource cards to buy a structure,
    or when a bank runs out of resources.

    Attributes:
        See Exception.

    Args:
        trading_entity (TradingEntity): The entity that lacked resources.

        resource_type (ResourceType or list of ResourceType): The type(s) of
          resource(s) the entity lacked.
    """

    def __init__(self, trading_entity, resource_types):

        resource_type_strs = map(
            lambda resource_type: str(resource_type),
            Utils.convert_to_list(resource_types)
        )

        resource_type_str = ’’

        if len(resource_type_strs) == 1:
            resource_type_str = resource_type_strs[0]
        else:
            resource_type_str = ’, ’.join(resource_type_strs[:-1]) +\
                ’, or ’ + resource_type_strs[-1]

        self.msg = ’{0} does not have enough {1} cards!’.format(
            trading_entity.__class__.__name__, resource_type_str)


class NotEnoughStructuresException(UserMessageException):
    """Raise when a player tries to build a structure despite having none left.

    Args:
        player (Player): The player that tried to build a structure.

        structure_name (str): The string name of structure the player attempted
          to build despite having run out.
    """

    def __init__(self, player, structure_name):
        self.msg = ’{0} does not have a {1} in stock.’.format(
            player.name, structure_name)


class NotEnoughDevelopmentCardsException(UserMessageException):
    """Raise when a player tries to buy a development card when none left."""

    def __init__(self):
        self.msg = ’No development cards remaining.’


class InvalidBaseStructureException(UserMessageException):
    """Raise when one tries to build an invalid upgrade or extension structure.

    Upgrade and extension structures need to be built off an appropriate base
    structure of a predetermined class. If the wrong class base structure is
    attempted, we should raise this error.
    """

    def __init__(self, base_structure, augmenting_structure):
        augments = augmenting_structure.augments()

        if augments is None:
            augments = ’an empty position’

        self.msg = ’{} must replace {}, but tried to replace a {}!’.format(
            augmenting_structure.name, augments, base_structure.name)


class BoardPositionOccupiedException(UserMessageException):
    """Raise when a player tries to build on a taken board position.

    Players can not place structures on positions taken by other players.
    Players can not replace existing structures with non-augmenting structures.
    """

    def __init__(self, position, structure, owning_player):

        self.msg = ’Position {} already has a {} belonging to {}.’.format(
            position, structure.name, owning_player.name)


class NoConfigValueDefinedException(UserMessageException):

    def __init__(self, dot_notation_str):

        self.msg = ’No config value defined for {}.’.format(dot_notation_str)


class NoSuchVertexException(UserMessageException):

    def __init__(self, tile, vertex_dir):

        self.msg = ’Tile has no vertex: {}’.format(vertex_dir)

class NoSuchEdgeException(UserMessageException):

    def __init__(self, tile, edge_dir):

        self.msg = ’Tile has no edge: {}’.format(edge_dir)


class InvalidStructurePlacementException(UserMessageException):
    """Raise when a player tries to place a structure somewhere they shouldn’t.

    E.g. no neighboring claimed roads, too close to another structure, etc.
    """

    def __init__(self):
        self.msg = ’Not a valid position to place the structure.’
```

```
import pdb
from engine.src.config.config import Config
from engine.src.lib.utils import Utils
from engine.src.exceptions import *
from engine.src.player import Player
from engine.src.dice import Dice
from engine.src.trading.trade_offer import TradeOffer
from engine.src.input_manager import InputManager
from engine.src.board.game_board import GameBoard
from engine.src.resource_type import ResourceType
from engine.src.position_type import PositionType
from engine.src.structure.structure import Structure
from engine.src.calamity.robber import Robber
from engine.src.longest_road_search import LongestRoadSearch

from imperative_parser.oracle import ORACLE

class Game(object):
    """A game of Settlers of Catan."""

    def __init__(self):

        Config.init()
        ORACLE.set(’game’, self)

        self.dice = Dice()
        self.board = GameBoard(Config.get(’game.board.radius’))
        ORACLE.set(’board’, self.board)

        # Place the robber on a fallow tile.
        self.robber = Robber()
        tile = self.board.get_tile_of_resource_type(ResourceType.FALLOW)
        tile.add_calamity(self.robber)

        self.players = []
        self.input_manager = InputManager

    def start(self):
        self.create_players()
        self.initial_settlement_and_road_placement()
        self.game_loop()

    def game_loop(self):

        max_point_count = 0

        while max_point_count < Config.get(’game.points_to_win’):
            for player in self.players:
                ORACLE.set(’player’, player)
                InputManager(self, player).cmdloop()
                self.update_point_counts()
                max_point_count = self.get_winning_player().get_total_points()

        # Print out game over message.
        winner = self.get_winning_player()
        print ’Game over. {0} wins with {1} points!\n’\
            .format(winner.name, winner.get_total_points())

    def create_players(self):
        """Create a new batch of players."""

        self.players = []
        player_names = InputManager.get_player_names()

        for player_name in player_names:
            self.players.append(Player(player_name))

        ORACLE.set(’players’, self.players)

    def place_structure(self, player, structure_name, must_border_claimed_edge=True,
                        struct_x=None, struct_y=None, struct_vertex_dir=None, free_to_build=False):
        """Place an edge or vertex structure.

        Prompts for placement information and attempts to place on board. Does
        not do any exception handling.
        """

        try:

            structure = player.get_structure(structure_name)

            if not free_to_build:
                # Requesting structure, not further resources
                trade_offer = TradeOffer(structure.cost, {})
                obstructing_entity, obstructing_resource_type = \
                    trade_offer.validate(player, self.board.bank)

                if not obstructing_entity and not obstructing_resource_type:
                    trade_offer.execute(player, self.board.bank)
                else:
                    raise NotEnoughResourcesException(obstructing_entity, obstructing_resource_type)

            if structure.position_type == PositionType.EDGE:
                prompt_func = InputManager.prompt_edge_placement
                placement_func = self.board.place_edge_structure
            elif structure.position_type == PositionType.VERTEX:
                prompt_func = InputManager.prompt_vertex_placement
                placement_func = self.board.place_vertex_structure

            x, y, struct_dir = prompt_func(self)

            params = [x, y, struct_dir, structure, must_border_claimed_edge]

            if struct_vertex_dir is not None:
                params.extend([struct_x, struct_y, struct_vertex_dir])

            placement_func(*params)

            player = structure.owning_player

            # Allocate points
            if structure.augments():
                # TODO: conversions from camelcase to underscore
                points = structure.point_value - Config.get(’game.structure.player_built.’ + structure_name.lower()).point_value
            else:
                points = structure.point_value

            player.points += points

            return x, y, struct_dir, structure

        except (NotEnoughStructuresException, NotEnoughResourcesException), e:
            raise
        except (BoardPositionOccupiedException, InvalidBaseStructureException,
                InvalidStructurePlacementException), e:

            if not free_to_build:
                # If we bought the structure but didn’t place it properly,
                # return the cost of the structure to the player.
                player.deposit_multiple_resources(structure.cost)

            # And return the structure to their storage.
            player.restore_structure(structure_name)

            # Raise the caught error so that callers of this method can handle
            # it in a custom fashion.
            raise

    def place_init_structure(self, player, structure_name,
                             must_border_claimed_edge=False,
                             struct_x=None, struct_y=None,
                             struct_vertex_dir=None):

        valid = False

        while not valid:
            try:
                free_to_build = True

                x, y, struct_dir, struct = self.place_structure(player, structure_name, must_border_claimed_edge,
                                        struct_x, struct_y, struct_vertex_dir, free_to_build)

                valid = True
            except (BoardPositionOccupiedException,
                    InvalidBaseStructureException,
                    InvalidStructurePlacementException), e:
                player.restore_structure(structure_name)
                InputManager.output(e)

        return x, y, struct_dir

    def initial_settlement_and_road_placement(self):

        InputManager.announce_initial_structure_placement_stage()

        for player in self.players:

            InputManager.announce_player_turn(player)

            # Place settlement
            InputManager.announce_structure_placement(player, 'Settlement')
            x, y, vertex_dir = self.place_init_structure(player, 'Settlement')

            # Place road
            InputManager.announce_structure_placement(player, 'Road')
            self.place_init_structure(player, 'Road', False, x, y, vertex_dir)

        distributions = Utils.nested_dict()

        for player in list(reversed(self.players)):

            InputManager.announce_player_turn(player)

            # Place settlement
            InputManager.announce_structure_placement(player, 'Settlement')
            x, y, vertex_dir = self.place_init_structure(player, 'Settlement')

            # Place road
            InputManager.announce_structure_placement(player, 'Road')
            self.place_init_structure(player, 'Road', False, x, y, vertex_dir)

            neighboring_tiles = filter(
                bool, self.board.get_adjacent_tiles_to_vertex(x, y, vertex_dir))

            # Give initial resource cards
            resource_types = filter(
                lambda resource_type: resource_type != ResourceType.FALLOW,
                map(lambda tile: tile.resource_type, neighboring_tiles)
            )

            for resource_type in resource_types:

                if not distributions[player][resource_type]:
                    distributions[player][resource_type] = 0

                distributions[player][resource_type] += \
                    Config.get('game.structure.player_built.settlement.base_yield')

        self.board.distribute_resources(distributions)
        InputManager.announce_resource_distributions(distributions)

    def roll_dice(self, value=None):

        roll_value = self.dice.roll()
        InputManager.announce_roll_value(roll_value)
        ORACLE.set('dice_value', roll_value)

        # If a calamity value, handle calamity
        distributions = self.board.distribute_resources_for_roll(roll_value)

        InputManager.announce_resource_distributions(distributions)

    def get_winning_player(self):
        """Get the player who is winning this game of Settlers of Catan."""

        return max(self.players, key=lambda player: player.points)

    def update_point_counts(self):

        for player in self.players:
            player.special_points = 0

        player_with_largest_army = max(self.players, key=lambda player: player.knights)


        # TODO: Move thresholds to config
        if player_with_largest_army.knights >= 3:
            print 'Largest army given to: {}'.format(player_with_largest_army)
            player_with_largest_army.special_points += 2

        player_road_len_dict = LongestRoadSearch(self.board).execute()

        for player, road_len in player_road_len_dict.iteritems():
            player.longest_road_length = road_len

        player_with_longest_road = max(player_road_len_dict)

        if player_with_longest_road.longest_road_length >= 5:
            print 'Longest road given to: {}'.format(player_with_longest_road)
            player_with_longest_road.special_points += 2
```

```python
import cmd
import sys
import pdb

from engine.src.config.config import Config
from engine.src.direction.vertex_direction import VertexDirection
from engine.src.direction.edge_direction import EdgeDirection
from engine.src.resource_type import ResourceType
from engine.src.vertex import Vertex
from engine.src.edge import Edge
from engine.src.exceptions import *
from engine.src.trading.trade_offer import TradeOffer
from engine.src.structure.structure import Structure


class InputManager(cmd.Cmd):
    """Class managing input for a given player’s turn. See docs for cmd.Cmd.

    Args:
        game (Game): The game being played.
        player (Player): Current player.

    Note that method docstrings are displayed to the user when they enter help.
    Implementation documentation should thus be given below the usual docstring.
    TODO: Commands do not support cancellation part way through.
    """
    def __init__(self, game, player):

        cmd.Cmd.__init__(self)

        self.game = game
        self.player = player
        self.prompt = ’> {0}: ’.format(self.player.name)

        self.has_rolled = False
        self.has_played_card = False

        self.structure_names = Utils.pluck(Config.get(’game.structure.player_built’), ’name’)

    def emptyline(self, line=None):
        """Override default emptyline behavior, which repeats last command."""
        if line is None:
            return
        self.default(line)

    def default(self, line):
        """Print menu of commands when unrecognized command given."""

        print ’Unrecognized command <{0}> given.’.format(line)
        self.do_help(None)

    def preloop(self):
        """Announce start of player turn."""

        msg = "{0}’s turn: ".format(self.player.name)
        InputManager.output(msg)

    def postloop(self):
        """Announce end of player turn."""

        msg = "End of {0}’s turn.".format(self.player.name)
        InputManager.output(msg)

    def do_debug(self, line):
        pdb.set_trace()

    def do_roll(self, value):
        """Roll the dice."""

        self.game.roll_dice(value)
        self.has_rolled = True

    # TODO: Move core logic to game.
    def do_trade_player(self, line):
        """Trade resources with other players."""

        if not self.has_rolled:
            print ’You must roll before you can trade.’
            return
        else:

            # Get list of requested resources
            msg = "Please enter a comma separated list of the number(s) " + \
                  "of the resource(s) you would like to offer."

            # offered_resources => resource_type => count
            offered_resources = InputManager.prompt_select_list_subset(
                msg, ResourceType.get_arable_types(),
                self.player.validate_resources
            )

            # Take csv list of offered resources
            msg = "Please enter a comma separated list of the number(s) " + \
                  "of the resource(s) you would like to receive."

            # requested_resources => resource_type => count
            requested_resources = InputManager.prompt_select_list_subset(
                msg, ResourceType.get_arable_types())

            # Create a trade offer
            trade_offer = TradeOffer(offered_resources, requested_resources)

            # Get player who will give requested resources and receive
            # offered resources.
            msg = "Please enter the number (e.g. ’1’) of the player " + \
                  "you would like to trade with."

            tradeable_players = filter(lambda player: player != self.player,
                                       self.game.players)

            if not tradeable_players:
                msg = ’No players to trade with.’
                InputManager.output(msg)
                return

            other_player = InputManager.prompt_select_list_value(
                msg, map(lambda player: player.name, tradeable_players),
                tradeable_players
            )

            try:
                other_player.trade(self.player, trade_offer)

                distributions = {
                    self.player: requested_resources,
                    other_player: offered_resources
                }

                InputManager.announce_trade_completed(trade_offer)
            # TODO: Specify explicit possible exceptions.
            except Exception as e:
                InputManager.output(e)

    def do_trade_bank(self, line):
        """Trade resources with the bank"""

        if not self.has_rolled:
            print ’You must roll before you can trade.’
            return
        else:
            # Get list of requested resources
            msg_offer = "Please enter the number of the resource you want to offer." + \
                "The bank buys 4 of a given resource, and returns 1 of any other resource."

            offered_resource_type = InputManager.prompt_select_list_value(
                msg_offer, ResourceType.get_arable_types()
            )

            msg_request = "Please enter the number of the resource you want to request."

            requested_resource_type = InputManager.prompt_select_list_value(
                msg_request, ResourceType.get_arable_types())

            offered_resources = {offered_resource_type: 4}
            requested_resources = {requested_resource_type: 1}

            trade_offer = TradeOffer(offered_resources, requested_resources)

            try:
                self.game.board.bank.trade(self.player, trade_offer)
                InputManager.announce_trade_completed(trade_offer)

            # TODO: Specify explicit possible exceptions.
            except Exception as e:
                InputManager.output(e)



    # TODO
    # TODO: long term. Refactor to be compatible w/ any trade intermediary.
    def do_trade_harbor(self, line):
        """Trade resources with a harbor."""
        print(’not yet implemented’)

    def do_build(self, line):
        """Build structures, including settlements, cities, and roads."""

        if not self.has_rolled:
            print ’You must roll before you can build.’
            return

        try:
            msg = "Please enter the number (e.g. ’1’) of the structure " + \
                  "you would like to build."

            structure_name = InputManager.prompt_select_list_value(
                msg, self.structure_names)

            self.game.place_structure(self.player, structure_name)

            self.game.update_point_counts()

        except (NotEnoughStructuresException, NotEnoughResourcesException,
                BoardPositionOccupiedException, InvalidBaseStructureException,
                InvalidStructurePlacementException), e:
            InputManager.output(e)

    # TODO: Enforce can’t play card bought during same turn.
    def do_buy_card(self, line):
        """Buy a development card."""

        if not self.has_rolled:
            msg = ’You must roll before you can buy a development card.’
            InputManager.output(msg)
        elif self.has_played_card:
            msg = ’You may only play one card per turn.’
            InputManager.output(msg)
        else:

            try:
                dev_card = self.game.board.bank.buy_development_card(self.player)
                dev_card.draw_card()

                success_msg = ’You received a {0}!’.format(str(dev_card))

                InputManager.input_default(success_msg, None, False)

            except (NotEnoughDevelopmentCardsException, NotEnoughResourcesException) as e:
                InputManager.output(e)

    def do_play_card(self, line):
        """Play a development card."""

        if self.has_played_card:
            msg = ’You may only play one card per turn.’
            InputManager.output(msg)
        else:

            msg = "Please enter the number (e.g. ’1’) of the development " + \
                  "card you would like to play."

            dev_card = InputManager.prompt_select_list_value(
                msg,
                map(lambda card: card.name, self.player.get_unplayed_development_cards()),
                self.player.get_unplayed_development_cards()
            )

            if not dev_card:
                InputManager.input_default(
                    ’Player has no development cards to choose from’,
                    None, False)
                return

            try:
                dev_card.play_card()
                self.game.update_point_counts()

            # TODO: Make clear which exceptions can be caught.
            except Exception as e:
                InputManager.output(e)

    # TODO: Improve.
    def do_print_board(self, line):
        """View the board."""

        for tile in self.game.board.iter_tiles():
            print tile

    def do_view_points(self, line):
        """View points per player (not including other players' hidden points)."""

        msg = 'Player Point Counts:\n'

        for player in self.game.players:
            points = player.get_total_points() if player == self.player \
                else player.get_visible_points()
            msg += '{}:\t{}'.format(player, points)

        InputManager.output(msg)

    def do_view_resources(self, line):
        """View your resource cards."""

        msg = '\n' + '\n'.join(map(
            lambda resource_type: '{}:\t{}'.format(resource_type, self.player.resources[resource_type]),
            self.player.resources
        ))

        InputManager.output(msg)

    # TODO
    def do_view_structures(self, line):
        """View your vertex and edge structures."""

        edge_structures = []
        vertex_structures = []

        for x, y in self.game.board.iter_tile_coords():
            tile = self.game.board.get_tile_with_coords(x, y)

            if not tile:
                continue

            for edge_dir in EdgeDirection:
                edge_val = tile.get_edge(edge_dir)

                if isinstance(edge_val, Structure) and \
                              edge_val.owning_player == self.player:

                    edge_structures.append( (tile, edge_dir, edge_val) )

            for vertex_dir in VertexDirection:
                vertex_val = tile.get_vertex(vertex_dir)

                if isinstance(vertex_val, Structure) and \
                              vertex_val.owning_player == self.player:
                    vertex_structures.append( (tile, vertex_dir, vertex_val) )

        structures = []
        tups_to_print = []

        for s in edge_structures:
            if s[2] not in structures:
                structures.append(s[2])
                tups_to_print.append(s)

        for s in vertex_structures:
            if s[2] not in structures:
                structures.append(s[2])
                tups_to_print.append(s)

        msg = '\n' + '\n'.join(map(lambda tup: 'Tile: {}\tDirection: {}\tStructure: {}'.format(
            tup[0], tup[1], tup[2].name), tups_to_print))

        InputManager.output(msg)

    def do_end_turn(self, line):
        """End your current turn."""

        if not self.has_rolled:
            print 'You must roll before you can end your turn.'
        else:
            return True

    def do_quit(self, line):
        """Quit the game for all players."""
        print '\nYou quit the game.'
        sys.exit(0)

    # Testing Methods
    def do_aybabtu(self, count):
        """All your base are belong to us."""

        if not count:
            count = 100
        else:
            count = int(count)

        for resource_type in ResourceType.get_arable_types():
            self.player.deposit_resources(resource_type, count)

    @staticmethod
    def output(msg):
        """Outputs the given message."""
        InputManager.input_default(msg, None, False)

    @staticmethod
    def input_default(msg, default=None, read_result=True):
        """Asks for user data using the format specified below.

        Returns:
            str. string entered by the user, or default if nothing was entered.
        """

        prompt = '> {0}'.format(str(msg))

        if default:
            prompt += " (or press enter to use default {0}): ".format(default)

        if read_result:
            prompt += '\n< '
            result = raw_input(prompt)
            # TODO: only return default if default flag true
            return result if result else default
        else:
            print prompt

    @staticmethod
    def get_player_names():
        """Prompts for and takes in player names.

        Returns:
            list. Of player name strings.
        """
        player_names = []
        num_players = 0

        while num_players <= 0:
            try:
                num_players = int(
                    InputManager.input_default(
                        'Enter number of players',
                        Config.get('game.player_count')
                    )
                )

                if num_players <= 0:
                    raise ValueError

            except ValueError:
                msg = 'Invalid number of players. Number must be an integer' + \
                    ' greater than zero.'
                InputManager.output(msg)

        # Shift range by 1 so prompts starting with player 1, not player 0
        for i in range(1, num_players + 1):
            msg = "Specify player {0}'s name".format(i)
            default = 'p{0}'.format(i)
            player_name = InputManager.input_default(msg, default)
            player_names.append(player_name)

        return player_names

    @staticmethod
    def prompt_select_player(game, players=None):

        if players is None:
            players = game.players

        msg = "Please enter the number (e.g. '1') of the player" + \
              "you would like to choose."

        return InputManager.prompt_select_list_value(msg, players)

    @staticmethod
    def prompt_tile_coordinates(game):

        x, y = None, None

        valid_coords = False

        while not valid_coords:
            try:
                x = int(InputManager.input_default(
                    'Please specify a tile x coordinate:', None))

                y = int(InputManager.input_default(
                    'Please specify a tile y coordinate:', None))

                valid_coords = game.board.valid_tile_coords(x, y)

                if not valid_coords:
                    raise ValueError
            except Exception:
                error_msg = "Invalid coordinates. Please try again."
                InputManager.output(error_msg)

        return x, y

    @staticmethod
    def prompt_select_list_value(prompt_msg, display_list, value_list=None):
        """Select and return a list element.

        Whenever we want to display a list and have the user select one entry
        in the list, we should use this method.

        If we want to display elements of one list to the user, but want to
        return a value different from the display value, we can provide both
        display and value lists. The user will select an index based on the
        values displayed, but the return value will result from using that same
        index to index into the value list.
        """

        if len(display_list) == 0:
            return None

        selected_element = None

        if value_list is None:
            value_list = display_list

        valid = False

        while not valid:

            for index, element in enumerate(display_list):
                print '({0}) {1}'.format(index + 1, element)

            try:
                index = int(InputManager.input_default(prompt_msg))

                if index < 1:
                    raise ValueError

                selected_element = value_list[index - 1]

                valid = True

            except (IndexError, ValueError, TypeError):
                msg = "Invalid number given. You must give a number " + \
                      "between 1 and {0}.".format(len(display_list))
                InputManager.output(msg)

        return selected_element

    @staticmethod
    def prompt_select_list_subset(prompt_msg, allowed_values_lst,
                                    validate_func=None):
        """Prompt user to select a subset of the allowed values list.

        User should input comma separated value list, where each value is an
        index of one of the displayed list elements.
        """

        selected_elements = []

        # Show the list of elements; indices offset by one for user readability.
        for index, element in enumerate(allowed_values_lst):
            print '({0}) {1}'.format(index + 1, element)

        valid = False
        index_list = []

        while not valid:

            index_list = InputManager.input_default(prompt_msg)\
                .replace(' ', '').split(',')

            try:

                resource_count_dict = Utils.convert_list_to_count_dict(map(
                    lambda index: allowed_values_lst[int(index) - 1],
                    index_list
                ))

                valid = validate_func(resource_count_dict) \
                    if validate_func is not None else True

            except (IndexError, ValueError):
                msg = "Invalid number given. All numbers must be " + \
                      "between 1 and {0}.".format(len(allowed_values_lst))
                InputManager.output(msg)
            except NotEnoughResourcesException as n:
                InputManager.output(n)

        return resource_count_dict

    @staticmethod
    def prompt_select_resource_type():

        msg = "Please enter the number (e.g. ’1’) of the resource type" + \
              "you would like to choose."

        return InputManager.prompt_select_list_value(msg, list(ResourceType))

    @staticmethod
    def prompt_vertex_direction():

        msg = "Please enter the number (e.g. ’1’) of the direction " + \
              "from the center of the tile to the vertex you would " + \
              "like to place a structure on."

        return InputManager.prompt_select_list_value(msg, list(VertexDirection))

    @staticmethod
    def prompt_edge_direction():

        msg = "Please enter the number (e.g. ’1’) of the direction " + \
              "from the center of the tile to the edge you would " + \
              "like to place a structure on."

        return InputManager.prompt_select_list_value(msg, list(EdgeDirection))

    @staticmethod
    def prompt_vertex_placement(game):

        x, y = InputManager.prompt_tile_coordinates(game)

        vertex_dir = InputManager.prompt_vertex_direction()

        return x, y, vertex_dir

    @staticmethod
    def prompt_edge_placement(game):

        x, y = InputManager.prompt_tile_coordinates(game)

        edge_dir = InputManager.prompt_edge_direction()

        return x, y, edge_dir

    # TODO: Roll announce methods into single method? Or programatically set.

    @staticmethod
    def announce_roll_value(roll_value):

        prompt = ’Player rolled a {0}’.format(roll_value)
        InputManager.output(prompt)

    @staticmethod
    def announce_initial_structure_placement_stage():

        prompt = ’Beginning initial structure placement stage.’
        InputManager.output(prompt)

    @staticmethod
    def announce_player_turn(player):

        prompt = "Beginning {0}’s turn.".format(player.name)
        InputManager.output(prompt)

    @staticmethod
    def announce_structure_placement(player, structure_name):

        prompt = "{0}, select where you would like to place your {1}".format(
            player.name, structure_name
        )
        InputManager.output(prompt)

    @staticmethod
    def announce_development_card_played(player, development_card):

        prompt = "{0} played a development card: {1}".format(
            player.name, str(development_card))
        InputManager.output(prompt)

    @staticmethod
    def announce_resource_distributions(distributions):

        msg = ’Distributing resources.’
        InputManager.output(msg)

        for player in distributions:
            for resource_type in distributions[player]:
                count = distributions[player][resource_type]

                if count:
                    msg = ’{0} received {1} {2} cards.’.format(
                        player.name, count, resource_type)
                    InputManager.output(msg)

    @staticmethod
    def announce_trade_completed(trade_offer):
        requested_resources = trade_offer.requested_resources
        offered_resources = trade_offer.offered_resources

        def generate_resources_readable_str(resources):
            return ", ".join(map(
                lambda res: str(resources[res]) + " " + str(res) + "(s)",
                (res for res in resources if resources[res] != 0)
            ))

        msg = "Trade completed. You bought " + \
            generate_resources_readable_str(requested_resources) + " and sold " + \
            generate_resources_readable_str(offered_resources) + "."

        InputManager.output(msg)
```

```python
# -*- coding: utf-8 -*-
import collections
from types import MethodType

def noop(cls, *args, **kwargs):
    pass

class Utils(object):
    """A general utility class."""

    @classmethod
    def init_from_dict(cls, obj, dct):

        for key, val in dct.iteritems():
            if Utils.is_function(val):
                setattr(obj, key, MethodType(val, obj, obj.__class__))
            else:
                setattr(obj, key, val)

    @classmethod
    def pluck(cls, dct, prop, do_filter=False):
        """Gets a list of values for the given property.

        Assumes the dct has key-value pairs where values are also dcts. Gets
        a list of values for the given property by taking them off each such
        value dct.
        """

        lst = []

        try:
            lst = map(lambda key: dct[key][prop], dct)

            if do_filter:
                lst = filter(lambda value: value is not None, lst)

        except KeyError:
            lst = []

        return lst

    @classmethod
    def remove_duplicates(cls, lst):

        result = []

        for e in lst:
            if e not in result:
                result.append(e)

        return result

    @classmethod
    def is_function(cls, func):
        return hasattr(func, ’__call__’)

    @classmethod
    def is_list(cls, lst):
        return hasattr(lst,"__iter__")

    @classmethod
    def noop(cls, *args, **kwargs):
        pass

    @classmethod
    def flatten(cls, lst):
        """Flattens a 2D list of lists."""

        return [nested_elem for elem in lst for nested_elem in elem]

    @classmethod
    def nested_dict(cls):
        """A nested default dictionary.

        Dictionaries in Python can become cumbersome if you constantly have to
        check if a key exists in a dictionary before proceeding. Using this as
        a dict definition allows the user to define arbitrarily nested values
        in the dictionary. Undefined nested values will return a defaultdict
        that, when cast to a boolean, will return False.

        Usage:
            my_dict = Utils.nested_dict()
            my_dict[k1][k2][k3] = value

        Taken from:
            http://stackoverflow.com/questions/16724788/how-can-i-get-python-to-automatically-create-missing-key-value-pairs-in-a-dictio
        """
        return collections.defaultdict(cls.nested_dict)

    @classmethod
    def convert_list_to_count_dict(cls, lst):

        dct = {}

        for val in lst:
            if val in dct:
                dct[val] += 1
            else:
                dct[val] = 1

        return dct

    @classmethod
    def convert_to_list(cls, e):
        """Convert to a list if not already a list."""
        return [e] if not Utils.is_list(e) else e

    @classmethod
    def dict_to_list(cls, dct):
        """Convert a counter-like dict to a list."""
        return Utils.flatten(map(lambda k: [k] * dct[k], dct))

    @classmethod
    def convert_format(cls, str):
        return str.replace(’-’, ’_’)

```

```
import pdb
from engine.src.lib.utils import Utils
from engine.src.direction.edge_direction import EdgeDirection
from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping
from engine.src.structure.structure import Structure
from engine.src.tile.hex_tile import HexTile


global vertices
global edges


def reset_metas():
    global vertices
    global edges
    vertices = Utils.nested_dict()
    edges = Utils.nested_dict()


def find_edge_meta(board, x, y, edge_dir):
    edge = edges[x][y][edge_dir]

    if not edge:
        tile = board.get_tile_with_coords(x, y)
        if tile:
            edge = EdgeMeta(board, x, y, edge_dir)
        else:
            edge = None

    return edge


def find_vertex_meta(board, x, y, vertex_dir):
    vertex = vertices[x][y][vertex_dir]

    if not vertex:
        tile = board.get_tile_with_coords(x, y)
        if tile:
            vertex = VertexMeta(board, x, y, vertex_dir)
        else:
            vertex = None

    return vertex


class VertexMeta(object):

    def __init__(self, board, x, y, vertex_dir):

        vertices[x][y][vertex_dir] = self

        self.board = board

        self.x = x
        self.y = y
        self.tile = self.board.get_tile_with_coords(self.x, self.y)

        self.vertex_dir = vertex_dir

        self.neighbors = []

        self.neighbors = self.find_neighbor_equivalents()

    def find_neighbor_equivalents(self):

        neighbors = []

        # Get the two edges of the found tile that have as an endpoint
        # a vertex of the given vertex direction.
        vertex_adj_edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(
            self.vertex_dir)

        for vertex_adj_edge_dir in vertex_adj_edge_dirs:
            neighbor_x = self.tile.x + vertex_adj_edge_dir[0]
            neighbor_y = self.tile.y + vertex_adj_edge_dir[1]
            neighbor_tile = self.board.get_neighboring_tile(self.tile, vertex_adj_edge_dir)

            # Edge tiles may not have neighboring tiles in the given direction.
            if neighbor_tile:
                neighbor_vertex_dir = HexTile.get_equivalent_vertex_dir(
                    self.vertex_dir, vertex_adj_edge_dir)

                neighbor = find_vertex_meta(self.board, neighbor_x, neighbor_y, neighbor_vertex_dir)

                neighbors.append(neighbor)

        return neighbors

    def __str__(self):
        return '({}, {}) {}'.format(self.x, self.y, self.vertex_dir)

    def __eq__(self, other):

        matches = self.x == other.x and \
                  self.y == other.y and \
                  self.vertex_dir == other.vertex_dir

        for neighbor in self.neighbors:
            matches = matches or \
                      neighbor.x == other.x and \
                      neighbor.y == other.y and \
                      neighbor.vertex_dir == other.vertex_dir

        return matches


class EdgeMeta(object):

    def __init__(self, board, x, y, edge_dir):

        edges[x][y][edge_dir] = self

        self.board = board

        self.x = x
        self.y = y
        self.tile = self.board.get_tile_with_coords(self.x, self.y)

        self.edge_dir = edge_dir
        self.edge_val = self.tile.get_edge(self.edge_dir)

        # Neighbor equivalent edge meta of same edge.
        self.neighbor_x = self.tile.x + self.edge_dir[0]
        self.neighbor_y = self.tile.y + self.edge_dir[1]
        self.neighbor_edge_dir = self.edge_dir.get_opposite_direction()

        edges[self.neighbor_x][self.neighbor_y][self.neighbor_edge_dir] = self

    def __str__(self):
        return '({}, {}) {}'.format(self.x, self.y, self.edge_dir)

    def __repr__(self):
        return ’({}, {}) {}’.format(self.x, self.y, self.edge_dir)

    def __eq__(self, other):

        matches_this = self.x == other.x and \
                       self.y == other.y and \
                       self.edge_dir == other.edge_dir

        matches_neighbor = self.neighbor_x == other.x and \
                           self.neighbor_y == other.y and \
                           self.neighbor_edge_dir == other.edge_dir

        return matches_this or matches_neighbor

class LongestRoadSearch(object):

    def __init__(self, board):
        self.board = board

    def execute(self):
        reset_metas()

        player_claimed_edges_dict = self.find_per_player_claimed_edges()
        player_road_len_dict = self.find_per_player_max_road_lengths(player_claimed_edges_dict)

        return player_road_len_dict

    def find_per_player_claimed_edges(self):

        player_claimed_edges_dict = Utils.nested_dict()
        checked_edges = Utils.nested_dict()

        for x, y in self.board.iter_tile_coords():
            tile = self.board.get_tile_with_coords(x, y)

            if not tile:
                continue

            for edge_dir in EdgeDirection:
                if not checked_edges[x][y][edge_dir]:
                    self.add_edge_to_dicts(x, y, edge_dir, player_claimed_edges_dict, checked_edges)

        return player_claimed_edges_dict

    def add_edge_to_dicts(self, x, y, edge_dir, player_claimed_edges_dict, checked_edges):

        edge_meta = find_edge_meta(self.board, x, y, edge_dir)

        if not edge_meta:
            checked_edges[x][y][edge_dir] = True
            return

        checked_edges[edge_meta.x][edge_meta.y][edge_meta.edge_dir] = True
        checked_edges[edge_meta.neighbor_x][edge_meta.neighbor_y][edge_meta.neighbor_edge_dir] = True

        if isinstance(edge_meta.edge_val, Structure):
            player = edge_meta.edge_val.owning_player

            if not player_claimed_edges_dict[player]:
                player_claimed_edges_dict[player] = []

            player_claimed_edges_dict[player].append(edge_meta)

    def find_per_player_max_road_lengths(self, player_claimed_edges_dict):

        player_road_len_dict = {}

        for player, player_claimed_edges in player_claimed_edges_dict.iteritems():
            player_road_len_dict[player] = self.find_max_road_len(player_claimed_edges)


        return player_road_len_dict

    def find_max_road_len(self, player_claimed_edges):
        """
        Args:
            player_claimed_edges (list): List of EdgeMetas.
        """

        max_road_len = 0

        for edge_meta in player_claimed_edges:
            edge_dir = edge_meta.edge_dir

            vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)

            remaining_edges = [e for e in player_claimed_edges if e != edge_meta]

            start_vertex = find_vertex_meta(self.board, edge_meta.x, edge_meta.y, vertex_dirs[0])
            end_vertex = find_vertex_meta(self.board, edge_meta.x, edge_meta.y, vertex_dirs[1])

            road_len = 1 + self.find_max_path_len(remaining_edges, end_vertex, edge_meta) \
                         + self.find_max_path_len(remaining_edges, start_vertex, edge_meta)

            if road_len > max_road_len:
                max_road_len = road_len

        return max_road_len

    def find_max_path_len(self, remaining_edges, end_vertex, edge_meta):

        neighbor_edge_metas = map(
            lambda edge_tuple: find_edge_meta(self.board, *edge_tuple),
            self.board.get_adjacent_edges(edge_meta.x, edge_meta.y, end_vertex.vertex_dir, False)
        )

        claimed_neighbors = [i for i in neighbor_edge_metas if i in remaining_edges]

        if claimed_neighbors:
            max_path_len = 0

            for claimed_neighbor in claimed_neighbors:
                remaining_edge_metas = [x for x in remaining_edges if (x != claimed_neighbor and x != edge_meta)]

                vertices = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(claimed_neighbor.edge_dir)

                vertex_metas = map(
                    lambda vertex_dir: find_vertex_meta(self.board, claimed_neighbor.x, claimed_neighbor.y, vertex_dir),
                    vertices
                )

                next_end_vertex = next(d for d in vertex_metas if d != end_vertex)

                path_len = 1 + self.find_max_path_len(remaining_edge_metas, next_end_vertex, claimed_neighbor)

                if path_len > max_path_len:
                    max_path_len = path_len

            return max_path_len
        else:
            return 0
```

```python
# -*- coding: utf-8 -*-
from engine.src.lib.utils import Utils
from engine.src.config.config import Config
from engine.src.structure.structure import Structure
from engine.src.trading.trading_entity import TradingEntity
from engine.src.exceptions import NotEnoughStructuresException


class Player(TradingEntity):
    """A player in a game of Settlers of Catan.

    Attributes:
        resources (dict): See TradingEntity.

        name (str): This player’s name.

    Args:
        name (str): Name to assign a new player.
    """

    def __init__(self, name):

        super(Player, self).__init__()

        self.name = name

        self.development_cards = []

        self.points = 0
        self.hidden_points = 0
        self.special_points = 0

        self.knights = 0
        self.longest_road_length = 0

        self.remaining_structure_counts = {}
        self.init_structure_counts()

    def __hash__(self):
        return hash(self.name)

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name

    def init_structure_counts(self):

        self.remaining_structure_counts = {}

        for structure in Config.get(’game.structure.player_built’).values():
            self.remaining_structure_counts[structure[’name’]] = structure[’count’]

    def get_total_points(self):
        return self.points + self.hidden_points + self.special_points

    def get_unplayed_development_cards(self):

        unplayed_dev_cards = filter(
            lambda dc: not dc.played, self.development_cards)

        return unplayed_dev_cards

    # TODO: pay for placing structure
    def get_structure(self, structure_name):
        """Get the given structure from the player’s stock, if any remains.

        Every time a player builds a structure, we need to remove from their
        stock, e.g. remaining_road_count etc. This method generalizes this
        process of removal for all structures.

        Args:
            structure_name (str): Class of structure to build.
        """

        structure_count = self.remaining_structure_counts[structure_name]

        if structure_count > 0:
            self.remaining_structure_counts[structure_name] -= 1

            # TODO: conversions between underscore and camel case
            config_path = ’game.structure.player_built.’ + structure_name.lower()
            structure_dict = Config.get(config_path)

            return Structure(self, **structure_dict)
        else:
            raise NotEnoughStructuresException(self, structure_name)

    # TODO: Restore cost of structure
    def restore_structure(self, structure_name):
        self.remaining_structure_counts[structure_name] += 1
```

```python
# -*- coding: utf-8 -*-
from enum import Enum


class PositionType(Enum):

    VERTEX = ’vertex’
    EDGE = ’edge’

    def __str__(self):
        return ’{0}’.format(self.value)

    def __eq__(self, other):
        return self.value == other

    @classmethod
    def find_by_value(cls, value):
        """Find the PositionType of the given value."""

        for position in cls:
            if value == position:
                return position
```

```python
# -*- coding: utf-8 -*-
import random
from enum import Enum


class ResourceType(Enum):
    """Defines the resource types available in a game of Settlers of Catan.

    Resources are produced by GameTile’s of the given resource type, and are
    used to build/buy structures, cards, etc.
    """

    # Arable tiles are non-fallow tiles.
    GRAIN = ’grain’
    LUMBER = ’lumber’
    WOOL = ’wool’
    ORE = ’ore’
    BRICK = ’brick’

    FALLOW = ’fallow’

    def __str__(self):
        return ’{0}’.format(self.value)

    def __eq__(self, other):
        return self.value == other

    @classmethod
    def get_priority_arable_types(cls):

        return cls.GRAIN, cls.LUMBER, cls.WOOL, cls.ORE, cls.BRICK

    @classmethod
    def get_arable_types(cls):
        """Get a list of non-fallow ResourceTypes only."""

        arable_types = filter(
            lambda resource_type: resource_type != ResourceType.FALLOW,
            list(ResourceType)
        )

        return arable_types

    @classmethod
    def iter_arable_types(cls):
        """Returns a generator over non-fallow enum members."""

        for resource_type in ResourceType.get_arable_types():
            yield resource_type

    @classmethod
    def random_arable_type(cls):
        """Return a random non-fallow ResourceType."""

        arable_types = ResourceType.get_arable_types()
        random_index = random.randint(0, len(arable_types))

        return arable_types[random_index]

    @classmethod
    def find_by_value(cls, value):
        """Find the ResourceType of the given value."""

        for resource in cls:
            if value == resource:
                return resource
```

```

```

```
# -*- coding: utf-8 -*-
from engine.src.config.config import Config
from engine.src.lib.utils import Utils

class Structure(object):
    """
    Attributes:
        owning_player
        name
        cost
        point_value
        extends
        upgrades
    """

    def __init__(self, owning_player, **kwargs):

        # Initialize default values.
        Config.init_from_config(self, ’game.structure.player_built.default’)

        # Overwrite default values with custom values.
        Utils.init_from_dict(self, kwargs)

        self.owning_player = owning_player

    def augments(self):
        if self.is_augmenting_structure():
            return self.upgrades if self.upgrades else self.extends
        return None

    def is_augmenting_structure(self):
        return self.extends or self.upgrades

    def __str__(self):
        return ’{} owned by {}’.format(self.name, self.owning_player)
```

```
# -*- coding: utf-8 -*-
from engine.src.tile.hex_tile import HexTile
from engine.src.resource_type import ResourceType
from engine.src.structure.structure import Structure


class GameTile(HexTile):
    """A hex tile as used in a game of Settlers of Catan.

    Args:
        resource (ResourceType): The resource/terrain of this hex.

        chit_value (int): The value of the chit (i.e. the circular number token)
          to be placed on this hex.

        calamities (list): A list of calamity objects placed on this tile i.e.
          whose passive effects currently affect this tile.
    """

    def __init__(self, x, y,
                 resource_type=ResourceType.FALLOW, chit_value=0):

        super(GameTile, self).__init__(x, y)

        self.resource_type = resource_type
        self.chit_value = chit_value
        self.calamities = []

    def __str__(self):
        return ’({0}, {1}) {2} {3}’.format(self.x, self.y,
                                           self.resource_type, self.chit_value)

    def __repr__(self):
        return self.__str__()

    def get_adjacent_vertex_structures(self):
        """Return any vertices that are structures."""

        return filter(
            lambda vertex: issubclass(vertex.__class__, Structure),
            list(self.iter_vertices())
        )

    def remove_calamity(self, calamity):
        """Remove a calamity from this tile.

        Args:
            calamity (Calamity): A calamity currently positioned on, and
              affecting, this tile, that will be removed.
        """

        self.calamities = filter(
            lambda existing_calamity: calamity != existing_calamity,
            self.calamities
        )

    def add_calamity(self, calamity):
        """Add a calamity to this tile.

        Args:
            calamity (Calamity): A calamity that, after calling this method,
              will be positioned on, and affect, this tile. The calamity to be
              added.

        Returns:
            boolean. Whether or not calamity was successfully added. Won’t be
              successfully added if had already been placed on this tile.
        """

        if calamity in self.calamities:
            return False
        else:
            self.calamities.append(calamity)
            return True

    def get_calamity_tile_placement_effects(self):
        """Get a list of tile placement effects for this tile’s calamities."""

        return filter(
            lambda effect: effect is not None,
            map(lambda calamity: calamity.tile_placement_effect,
                self.calamities)
        )
```

```
# -*- coding: utf-8 -*-

from engine.src.exceptions import *
from .tile import Tile
from engine.src.vertex import Vertex
from engine.src.edge import Edge
from engine.src.direction.vertex_direction import VertexDirection
from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping


class HexTile(Tile):
    """A hexagonal tile, with 6 edges and 6 vertices.

    Attributes:
        vertices (dict): The 6 vertices of this tile, indexed by the
          VertexDirection of the vertex i.e. the tuple of the direction,
          not its string name.

        edges (dict): The edges of this tile, indexed by a pair of vertex
          directions.
          Note that edges are undirected so edges[src][dst] = edges[dst][src].

    Args:
        x (int): The x-coordinate of this tile in the axial coordinate system
          used by the board to which this tile belongs.

        y (int): The y-coordinate of this tile in the axial coordinate system
          used by the board to which this tile belongs.

    TODO: x and y are mostly here for testing purposes. Removable.
    """

    def __init__(self, x, y):
        self.x = x
        self.y = y

        self.vertices = {}
        self.edges = {}
        self._create_vertices_and_edges()

    def __repr__(self):
        return ’({0}, {1})’.format(self.x, self.y)

    def __str__(self):
        return ’({0}, {1})’.format(self.x, self.y)

    def _create_vertices_and_edges(self):
        """Create brand new vertices and edges for this tile."""

        self.vertices = {}
        self.edges = {}

        for (start_vertex_dir, end_vertex_dir) in VertexDirection.pairs():

            end_vertex = Vertex()
            self.vertices[end_vertex_dir] = end_vertex

            self.add_edge(start_vertex_dir, end_vertex_dir)

    def add_edge(self, start_vertex_dir, end_vertex_dir, edge=Edge()):
        """Add an edge connecting vertices at given directions to this tile.

        Since edges aren’t directed, edges[src][dst] = edges[dst][src].

        Args:
            start_vertex_dir (VertexDirection): Direction relative to
              this tile to the vertex that comprises one end of the edge to add.

            end_vertex_dir (VertexDirection): Direction relative to
              this tile of the edge-to-add’s endpoint vertex.

        Returns:
            None.

        TODO: enforce that these are adjacent vertex directions.
        """

        if start_vertex_dir not in self.edges:
            self.edges[start_vertex_dir] = {}

        if end_vertex_dir not in self.edges:
            self.edges[end_vertex_dir] = {}

        self.edges[start_vertex_dir][end_vertex_dir] = edge
        self.edges[end_vertex_dir][start_vertex_dir] = edge

    def update_common_edge_and_vertices(self, edge_direction,
                                        neighboring_tile):
        """Update vertices and edges this tile shares with the neighboring tile.

        Args:
            edge_direction (EdgeDirection): The given neighboring tile
              should share an edge at the given direction relative to this tile.

            neighboring_tile (Tile): The tile whose relevant vertices and
              edges we should use to overwrite those of this tile.

        Returns:
            None.
        """
        # Get the directions of the vertices comprising the endpoints of the
        # edge in the given edge_direction i.e. the edge shared between this
        # tile and the neighbor tile.
        start_vertex_dir, end_vertex_dir = \
            EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_direction)

        # Get the symmetric directions for the neighbor tile.
        neighbor_start_vertex_dir, neighbor_end_vertex_dir = \
            EdgeVertexMapping.get_vertex_dirs_for_edge_dir(
                edge_direction.get_opposite_direction())

        # Get the vertices belonging to the neighboring tile at the found
        # directions.
        start_vertex = neighboring_tile.vertices[neighbor_start_vertex_dir]
        end_vertex = neighboring_tile.vertices[neighbor_end_vertex_dir]

        # Replace this tile’s vertices with the neighbor’s vertices.
        self.vertices[start_vertex_dir] = start_vertex
        self.vertices[end_vertex_dir] = end_vertex

        # Replace this tile’s edge with the neighbor’s edge.
        self.add_edge(start_vertex_dir, end_vertex_dir,
                      neighboring_tile.edges[start_vertex_dir][end_vertex_dir])

    def iter_edges(self):
        """Iterate over the edges of this tile."""

        for (start_vertex_dir, end_vertex_dir) in VertexDirection.pairs():
            yield self.vertices[start_vertex_dir][end_vertex_dir]

    def iter_vertices(self):
        """Iterate over the vertices of this tile."""

        for vertex_direction in VertexDirection:
            yield self.vertices[vertex_direction]

    def update_vertex(self, vertex_direction, vertex_value):
        """Update the vertex defined by the given vertex direction."""

        self.vertices[vertex_direction] = vertex_value

    @classmethod
    def get_equivalent_vertex_dir(cls, vertex_dir, edge_dir):
        """Get the equivalent vertex as the given one, relative to this tile.

        Consider two adjacent tiles, one of which we will think of as the
        base_tile, relative to which vertex_dir and edge_dir are defined,
        and its neighboring adj_tile. If we know the direction of a vertex
        relative to base_tile, and we want to find the direction to the same
        vertex relative to adj_tile, we should use this method.

        Args:
            vertex_dir (VertexDirection): See above.

            edge_dir (EdgeDirection): Edge direction of the shared edge,
              relative to the given tile, of the edge shared by base_tile and
              adj_tile, as described above.

        Returns
            VertexDirection.
        """

        # Get the vertex directions, relative to this tile, of the vertices
        # that comprise the endpoints of the given edge_dir. Since edge_dir is
        # relative to the base_tile, we must find it’s opposite to find the
        # edge_dir relative to this tile.
        opposite_edge_vertices = \
            EdgeVertexMapping.get_vertex_dirs_for_edge_dir(
                edge_dir.get_opposite_direction())

        # Filter out the vertex that is opposite the given vertex, since that
        # will not correspond to the same vertex relative to this tile.
        vertex = next(vertex for vertex in opposite_edge_vertices if
                      vertex != vertex_dir.get_opposite_direction())

        return vertex

    def get_vertex(self, vertex_dir):

        if vertex_dir in self.vertices:
            return self.vertices[vertex_dir]
        else:
            raise NoSuchVertexException(self, vertex_dir)

    def get_edge(self, edge_dir):

        vert_src_dir, vert_dst_dir = \
            EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)

        if vert_src_dir in self.edges:
            if vert_dst_dir in self.edges[vert_src_dir]:
                return self.edges[vert_src_dir][vert_dst_dir]

        raise NoSuchEdgeException(self, edge_dir)
```

```
# -*- coding: utf-8 -*-


class Tile(object):
    pass
```

```python
# -*- coding: utf-8 -*-
import random

from engine.src.config.config import Config
from engine.src.trading.trading_entity import TradingEntity
from engine.src.trading.trade_offer import TradeOffer
from engine.src.exceptions import *
from engine.src.card.development_card import DevelopmentCard


class Bank(TradingEntity):
    """Represents the bank of all available resource cards.

    Attributes:
        resources (dict): See TradingEntity.

        development_cards (list): A list of different development card objects.

    Args:
        tile_count (int): Number of tiles for the board this bank will be used
          with.
    """

    def __init__(self, tile_count=None):
        if tile_count is None:
            tile_count = Config.get('game.board.tile_count')

        super(Bank, self).__init__()

        self.development_cards = []

        self._default_init_development_cards()
        self._default_init_resources(tile_count)

    def _default_init_resources(self, tile_count):
        """Determine the initial resources for the bank.

        Though not officially a rule, one notices that the default card
        allocation for the base game is such that there is, for each resource
        type, the same number of cards as there are tiles on the board. In
        order to make this function work for different size boards, this is
        the rule used to default allocate resource types.

        Args:
            tile_count (int): Number of tiles on the playing board.

        Returns:
            None. Modifies self.resources.
        """

        super(Bank, self)._default_init_resources(tile_count)

    def _default_init_development_cards(self):
        """Add a configured number of each development card type to the bank."""

        dev_card_dict = Config.get('game.card.development')

        for name, card in dev_card_dict.iteritems():
            for _ in range(card['count']):
                dev_card = DevelopmentCard(**card)
                self.development_cards.append(dev_card)

        random.shuffle(self.development_cards)

    def buy_development_card(self, player):
        """Let the given player purchase a development card from the bank."""

        if not self.development_cards:
            raise NotEnoughDevelopmentCardsException

        card = self.development_cards.pop()

        # Create a trade offer where there are no requested resources,
        # just offered resources (cost of development card).
        trade_offer = TradeOffer(card.cost, {})

        obstructing_entity, obstructing_resource_type = \
            trade_offer.validate(player, self)

        # If the trade offer is valid, transfer the cost cards and give
        # the player the development card.
        if not obstructing_entity and not obstructing_resource_type:
            trade_offer.execute(player, self)
            player.development_cards.append(card)
            return card
        # Otherwise, return the development card to the deck.
        else:
            self.development_cards.append(card)
            raise NotEnoughResourcesException(obstructing_entity, obstructing_resource_type)
```

```python
# -*- coding: utf-8 -*-
from engine.src.trading.trading_intermediary import TradingIntermediary


class Harbor(TradingIntermediary):
    """Represents a trading harbor in Settlers of Catan.

    Attributes:
        supplier (TradingEntity): See TradingIntermediary.

        trade_criteria (TradeCriteria): A rule that must be followed for a
          trade conducted through this harbor to be considered valid.
    """

    def __init__(self, supplier, trade_criteria):

        super(Harbor, self).__init__(supplier)
        self.trade_criteria = trade_criteria

    def trade(self, other_entity, trade_offer):
        """Attempt to execute the trade only if it follows the trade criteria.

        Args:
            See TradingIntermediary for:
                other_entity (TradingEntity)
                trade_offer (TradeOffer)

        Returns:
            None.
        """

        if self.trade_criteria.permits(trade_offer):
            super(Harbor, self).trade(other_entity, trade_offer)
```

```
# -*- coding: utf-8 -*-
from enum import Enum
from engine.src.resource_type import ResourceType


class TradeOffer(object):
    # TODO: Convert resources to collections.Counter

    def __init__(self, offered_resources, requested_resources):

        self.requested_resources = TradeOffer._get_empty_resources()
        self.requested_resources.update(requested_resources)

        self.offered_resources = TradeOffer._get_empty_resources()
        self.offered_resources.update(offered_resources)

    @staticmethod
    def _get_empty_resources():

        resources = {}

        for arable_type in ResourceType.get_arable_types():
            resources[arable_type] = 0

        return resources

    def validate(self, proposing_entity, receiving_entity):
        """See if this trade can be carried out between the given entities.

        Args:
            proposing_entity (TradingEntity): The entity that proposed the
              trade, i.e. that wants to give the offered_resources and receive
              the requested_resources of this trade.

            receiving_entity (TradingEntity): The other entity to whom this
              trade was proposed and who will receive the offered_resources and
              give the requested_resources.

        Returns:
            TradingEntity, ResourceType. If the trade cannot be completed, this
              method returns the entity that is blocking it and the resource
              they lack. If the trade can be completed, it will return None.
        """

        # Check that the proposing_entity has all the resources listed in this
        # trade’s offered_resources dict.
        for resource_type, count in self.offered_resources.iteritems():
            if proposing_entity.resources[resource_type] < count:
                return proposing_entity, resource_type

        # Check that the receiving entity has all the resources listed in this
        # trade’s requested_resources dict.
        for resource_type, count in self.requested_resources.iteritems():
            if receiving_entity.resources[resource_type] < count:
                return receiving_entity, resource_type

        return None, None

    def execute(self, proposing_entity, receiving_entity):
        """Execute this trade based on the given trade entities.

        This call should always be preceded by a call to self.validate().

        Args:
            See self.validate()

        Returns:
            None.
        """

        # Take the offered resources from the entity that proposed the deal
        # and give them to the entity that accepted the deal.
        for resource_type, count in self.offered_resources.iteritems():
            proposing_entity.withdraw_resources(resource_type, count)
            receiving_entity.deposit_resources(resource_type, count)

        # Take the resources requested by the proposing entity from the
        # entity that accepted the deal and give them to the proposing entity.
        for resource_type, count in self.requested_resources.iteritems():
            proposing_entity.deposit_resources(resource_type, count)
            receiving_entity.withdraw_resources(resource_type, count)


class TradeMetaCriteria(Enum):
    ANY = 1
    SAME = 2


class TradeCriteria(TradeOffer):
    """Defines different trade criteria."""

    def __init__(self, offered_resources=None, requested_resources=None,
                 offered_meta=None, requested_meta=None):

        super(TradeCriteria, self).__init__(offered_resources,
                                            requested_resources)

        self.offered_meta = TradeCriteria._get_empty_meta()
        self.requested_meta = TradeCriteria._get_empty_meta()

        self.offered_meta.update(offered_meta)
        self.requested_meta.update(requested_meta)

    @staticmethod
    def _get_empty_meta():

        meta = {}

        for criteria in TradeMetaCriteria:
            meta[criteria] = 0

        return meta

    def permits(self, trade_offer):

        valid_offer = self.valid(self.offered_resources, self.offered_meta,
                                 trade_offer.offered_resource)

        valid_req = self.valid(self.requested_resources, self.requested_meta,
                               trade_offer.requested_resources)

        return valid_offer and valid_req

    @staticmethod
    def valid(crit_resources, crit_meta, offered_resources):

        offered_resources = offered_resources.copy()

        valid = True

        # First handle meta
        if valid and TradeMetaCriteria.SAME in crit_meta:

            valid = False

            req_same_resource_count = crit_meta[TradeMetaCriteria.SAME]

            for resource_type, count in offered_resources.iteritems():
                if count >= req_same_resource_count:
                    offered_resources[resource_type] -= req_same_resource_count
                    valid = True
                    break

        if valid and TradeMetaCriteria.ANY in crit_meta:

            req_any_resource_count = crit_meta[TradeMetaCriteria.ANY]

            for resource_type, count in offered_resources.iteritems():
                if count > 0:
                    deduct = min(count, req_any_resource_count)

                    req_any_resource_count -= deduct
                    offered_resources[resource_type] -= deduct

            if req_any_resource_count > 0:
                valid = False

        if valid:
            # Now handle normal resources
            for resource_type, count in crit_resources.iteritems():
                if count != offered_resources[resource_type]:
                    valid = False

        return valid
```

```python
# -*- coding: utf-8 -*-
import random
from collections import Counter
from engine.src.lib.utils import Utils
from engine.src.exceptions import NotEnoughResourcesException
from engine.src.resource_type import ResourceType
from engine.src.trading.trade_offer import TradeOffer


class TradingEntity(object):
    """Represents an entity capable of storing and trading resources.

    Attributes:
        resources (dict): Represents all resources currently owned by this
          entity. Keys are arable ResourceTypes and values are integers
          representing the amount of a particular resource type the entity has.

    TODO: This should be an abstract class.
    """

    def __init__(self):
        self.resources = {}
        # TODO: Freak error where Python isn’t recognizing default arg.
        self._default_init_resources(0)

    def _default_init_resources(self, count):
        """Initialize this entity to have count resources per resource type.

        Args:
            count (int): Number of each arable resource this entity will have.

        Returns:
            None. Modifies self.resources.
        """

        self.resources = {}
        for arable_type in ResourceType.get_arable_types():
            self.resources[arable_type] = count

    def count_resources(self):
        return sum(self.resources.values())

    def validate_resources(self, resources):
        """Check that this player has at least as many resources as given."""

        default_resources = TradeOffer._get_empty_resources()
        default_resources.update(resources)

        resources = default_resources

        # This entity does not have the given resources if the difference
        # between its count and the given resources dict count for any given
        # resource type is negative.
        resource_debt = {resource_type: count - resources[resource_type]
                         for resource_type, count in self.resources.items()
                         if count - resources[resource_type] < 0}

        valid = len(resource_debt.keys()) == 0

        if valid:
            return True
        else:
            raise NotEnoughResourcesException(self, resource_debt.keys())

    def get_resource_list(self):
        """Get a list of resource types, one for each "card" this player has."""

        return Utils.flatten(map(
            lambda resource_type:
                [resource_type] * self.resources[resource_type],
            self.resources
        ))

    def transfer_resources(self, to_entity, resource_type, resource_count):
        """Transfer specified resources from this entity to the given entity."""

        self.withdraw_resources(resource_type, resource_count)
        to_entity.deposit_resources(resource_type, resource_count)

    def withdraw_resources(self, resource_type, resource_count):
        """Withdraw the specified number of resources from the entity.

        Args:
            resource_type (ResourceType): Type of resource to withdraw.

            resource_count (int): Number of resources of the given type to
              withdraw.

        Raises:
            NotEnoughResourcesException. When the withdrawal is for more
              resources than the entity currently has.
        """

        if resource_type == ResourceType.FALLOW:
            # TODO: raise exception.
            return

        if self.resources[resource_type] >= resource_count:
            self.resources[resource_type] -= resource_count
        else:
            raise NotEnoughResourcesException(self, resource_type)

    def withdraw_random_resource(self):
        """Remove a random resource from this trading entity.

        Note that this method only withdraws a single random resource.
        Callers of this method should check to make sure that this entity
        still has resources using self.count_resources().
        """

        resources = self.get_resource_list()

        resource_type = random.choice(resources)

        self.resources[resource_type] -= 1

        return resource_type

    def deposit_multiple_resources(self, resource_type_count_dict):

        for resource_type, count in resource_type_count_dict.iteritems():
            self.deposit_resources(resource_type, count)

    def deposit_resources(self, resource_type, resource_count):
        """Deposit the specified number of resources from the entity.

        Args:
            resource_type (ResourceType): Type of resource to deposit.

            resource_count (int): Number of resources of the given type to
              deposit.
        """

        if resource_type != ResourceType.FALLOW:
            self.resources[resource_type] += resource_count

    def trade(self, requesting_entity, trade_offer):
        """Trade one resource for another at a given ratio.

        Args:
            requesting_entity (TradingEntity): Entity who has proposed a trade
              wherein they offer the trade’s offered_resources and request the
              trade’s requested_resources from this entity.

            trade (Trade): Keeps track of how many of which resource are being
              offered and requested.

        Raises:
            NotEnoughResourcesException. When this or the other entity lacks
              the resources to complete the trade.
        """

        obstructing_entity, obstructing_resource_type = \
            trade_offer.validate(requesting_entity, self)

        if obstructing_entity is not None:
            raise NotEnoughResourcesException(obstructing_entity,
                                              obstructing_resource_type)

        else:
            trade_offer.execute(requesting_entity, self)
```

```
# -*- coding: utf-8 -*-
from engine.src.trading.trading_entity import TradingEntity


class TradingIntermediary(object):
    """Represents an entity capable of trading resources on behalf of two other
    TradingEntity’s, but incapable of storing resources itself.

    Args:
        supplier (TradingEntity): The entity who owns the resources this
          intermediary is allowed to trade on its behalf.
    """

    def __init__(self, supplier):

        if not isinstance(supplier, TradingEntity):
            message = ’Invalid trading entity given as supplier’
            raise ValueError(message)

        self.supplier = supplier

    def trade(self, other_entity, trade_offer):
        """Attempt to execute the given trade.

        Args:
            other_entity (TradingEntity): Entity that proposed the trade to
              the harbor.

            trade_offer (TradeOffer): Trade offer crafted by the other entity.

        Returns:
            None.
        """

        self.supplier.trade(other_entity, trade_offer)
```

```
class AsciiHexBoard(object):

    board_string = \
"""
           / \     / \     / \
         /     \ /     \ /     \
        |  -2,2 | -1, 2 |  0,2  |
        |       |       |       |
       / \     / \     / \     / \
     /     \ /     \ /     \ /     \
    |  -2,1 | -1,1  |  0, 1 |  1,1  |
    |       |       |       |       |
   / \     / \     / \     / \     / \
 /     \ /     \ /     \ /     \ /     \
|  -2,0 | -1,0  |  0,0  |  1,0  |  2,0  |
|       |       |       |       |       |
 \     / \     / \     / \     / \     /
   \ /     \ /     \ /     \ /     \ /
    | -1,-1 | 0,-1  |  1,-1 |  2,-1 |
    |       |       |       |       |
     \     / \     / \     / \     /
       \ /     \ /     \ /     \ /
        |  0,-2 | 1,-2  |  2,-2 |
        |       |       |       |
         \     / \     / \     /
           \ /     \ /     \ /
"""
```

```python
# -*- coding: utf-8 -*-
from abc import ABCMeta


class Vertex(object):
    __metaclass__ = ABCMeta
```

```python
# TODO: Cleanup. Separate module registration with game run logic?

# Add engine package to Python path.
import sys
import os

sys.path.insert(1, os.path.dirname(os.path.dirname(os.path.abspath(__file__))))


# Catch SIGINT for prettier force quit handling.
import signal

def signal_handler(signal, frame):
    print '\nYou force quit the game.'
    sys.exit(0)

signal.signal(signal.SIGINT, signal_handler)


# Run main game loop.
from engine.src.game import Game
from engine.src.config.config import Config

print Config.get('game.board.tile_count')

# g = Game()
# g.start()

```

```
"""
A pretty-printing dump function for the ast module.  The code was copied from
the ast.dump function and modified slightly to pretty-print.

Alex Leone (acleone ˜AT˜ gmail.com), 2010-01-30
"""

from ast import *

def dump(node, annotate_fields=True, include_attributes=False, indent=’  ’):
    """
    Return a formatted dump of the tree in *node*.  This is mainly useful for
    debugging purposes.  The returned string will show the names and the values
    for fields.  This makes the code impossible to evaluate, so if evaluation is
    wanted *annotate_fields* must be set to False.  Attributes such as line
    numbers and column offsets are not dumped by default.  If this is wanted,
    *include_attributes* can be set to True.
    """
    def _format(node, level=0):
        if isinstance(node, AST):
            fields = [(a, _format(b, level)) for a, b in iter_fields(node)]
            if include_attributes and node._attributes:
                fields.extend([(a, _format(getattr(node, a), level))
                               for a in node._attributes])
            return ’’.join([
                node.__class__.__name__,
                ’(’,
                ’, ’.join((’%s=%s’ % field for field in fields)
                           if annotate_fields else
                           (b for a, b in fields)),
                ’)’])
        elif isinstance(node, list):
            lines = [’[’]
            lines.extend((indent * (level + 2) + _format(x, level + 2) + ’,’
                         for x in node))
            if len(lines) > 1:
                lines.append(indent * (level + 1) + ’]’)
            else:
                lines[-1] += ’]’
            return ’\n’.join(lines)
        return repr(node)
    if not isinstance(node, AST):
        raise TypeError(’expected AST, got %r’ % node.__class__.__name__)
    return _format(node)

if __name__ == ’__main__’:
    import sys
    for filename in sys.argv[1:]:
        print ’=’ * 50
        print ’AST tree for’, filename
        print ’=’ * 50
        f = open(filename, ’r’)
        fstr = f.read()
        f.close()
        print dump(parse(fstr, filename=filename), include_attributes=True)
        print
```

```
from collections import defaultdict

def get_registry():
    """Produces a registration decorator that allows methods to be gathered under tags
    """
    registry = defaultdict(list)
    def register(nonterminal):
        def registrar(func):
            registry[nonterminal] += [func]
            return func
        return registrar
    register.get = lambda x: registry[x]
    return register

def gen_grammar(name, nonterminals, indent=4):
    """Generates a grammar docstring for the provided name and nonterminals
    E.x. name : nonterminal1
              | nonterminal2

    Args:
        name (String): The nonterminal name
        nonterminals (List): A list of the nonterminals it’s associated with

    Returns:
        String. A docstring representing the grammar of the nonterminal
    """
    docstring = "{} : {}".format(name, nonterminals[0])
    padding = ’ ’ * (len(name) + 1 + indent) + ’| ’

    if len(nonterminals) > 1:
        docstring += ’\n’ + padding + (’\n’ + padding).join(nonterminals[1:])

    return docstring

def trivial(name, nonterminals, indent=4, suffix=’’):
    """Generates a method for a trivial terminal, where p[0] = p[1]

    Args:
        name (String): A string representing the nonterminal name
        nonterminals (List): A list of strings representing the nonterminals it’s linked to

    Named Args:
        indent (Int): 4 -- An int representing the amount of indentation in the file
        suffix (String): ’’ -- A string representing a suffix that should be added to the name of the function

    Returns:
        Func. A function with the provided name and a generated grammar docstring
    """
    def template(p):
        p[0] = p[1]

    template.__doc__ = gen_grammar(name, nonterminals, indent)

    template.__name__ = template.func_name = ’p_’ + name + suffix

    return template

def trivial_from_registry(name, registry, indent=4, suffix=’’):
    """Generates a method for a trivial terminal, where p[0] = p[1], sourcing nonterminals from a registry

    Args:
        name (String): A string representing the nonterminal name
        registry (Dict): A registry generated by the get_registry() function

    Named Args:
        indent (Int): 4 -- An int representing the amount of indentation in the file
        suffix (String): ’’ -- A string representing a suffix that should be added to the name of the function

    Returns:
        Func. A function with the provided name and a generated grammar docstring
    """
    return trivial(name, [func.__doc__.split(’:’)[0].strip() for func in registry.get(name)], indent=indent, suffix=suffix)
```

```python
#import sys
#sys.path.append(’..’)
#from ..engine.src.lib.utils import Utils
from collections import defaultdict

class StateNotFound(Exception):
    """Thrown when a dependency injection tries to inject a variable that isn’t part of the declared game state
    """
    pass


class GameOracle(object):
    """A wrapper object for the game state, providing a simple interface to isolate development of the imperative
    parser from the game engine
    """

    def __init__(self, state={}):
        """Creates an instance of a GameOracle

        Named Args:
            state (Dict): {} -- a dictionary containing references from variable name strings to game state objects

        Returns:
            GameOracle. An oracle which can access the provided state dictionary
        """
        self.game_state = state

    def get(self, name):
        """Get a variable from the GameOracle’s state

        Args:
            name (String): A string representing the name of the variable to retrieve

        Returns:
            Any. The value of the variable being retrieved

        Throws:
            StateNotFound -- when a state being accessed isn’t present in the state dict
        """
        try:
            return self.game_state[name]
        except KeyError:
            raise StateNotFound("Variable \"%s\" not present in game state" % name)

    def set(self, name, var):
        """Set a particular variable in the state dict to a particular value

        Args:
            name (String): A string representing the name to store the variable under
            var (Any): The value to store for the variable
        """
        self.game_state[name] = var

# Access game state through the game oracle
ORACLE = GameOracle(defaultdict(list))
```

```
import ast
from collections import defaultdict

import ply.lex as lex
import ply.yacc as yacc

from grammar_utils import get_registry, trivial_from_registry, trivial, gen_grammar
from utils import flatten, find_column

# Allow dependency injection using the predefined GameOracle
from oracle import ORACLE

class RewriteInjected(ast.NodeTransformer):
    def __init__(self, injected):
        """Creates a NodeTransformer object to replace calls to injected parameters with calls to a lookup table

        Args:
            injected (Iterable): An iterable representing the list of injected parameter names

        Returns:
            An instance of RewriteInjected whose visit method will rewrite the injected nodes
        """
        super(RewriteInjected, self).__init__()
        self.injected = set(injected)

    def visit_Name(self, node):
        if node.id in self.injected:
            return ast.copy_location(ast.Call(
                ast.Attribute(
                    ast.Name(’ORACLE’, ast.Load()),
                             ’get’, ast.Load()
                ), [ast.Str(node.id)], [], None, None), node)
        else:
            return self.generic_visit(node)

# Automatically build no-op nonterminals
register = get_registry()

def gen_function(name):
    """Generates a function for the given trivial nonterminal based on the registry

    Args:
        name (String): A string representing the nonterminal to generate the function for

    Returns:
        Func. A trivial function p[0] = p[1] for the nonterminal
    """
    return trivial_from_registry(name, register, suffix=’_reg’)

# Helper functions

def listify(p, item_pos=1, list_pos=3, size_check=2):
    """Creates a list of values from the given nonterminal parse p

    Args:
        p (List): A list representing the parse

    Named Args:
        item_pos (Int): 1 -- An int representing the position of the item at the head of the list
        list_pos (Int): 3 -- An int representing the position of the rest of the list
        size_check (Int): 2 -- An int representing the length of the parse of a single item of the list

    Returns:
        List. The parse p, with p[0] set to the list of items
    """
    p[0] = [p[item_pos]] if p[item_pos] else []
    if len(p) > size_check:
        p[0].extend(p[list_pos])
    return p

# Token declarations

# TODO allow reserved words in strings
reserved = {k: k.upper() for k in [
    ’func’,
    ’return’,
    ’print’,
    ’if’,
    ’else’,
    ’or’,
    ’and’,
    ’not’,
    ’while’,
    ’for’,
    ’to’
]}
tokens = [’ID’, ’NUM’, ’COMPOP’, ’AUGASSIGN’, ’NEWLINE’, ’IN’, ’STRING’] + list(reserved.values())
literals = [’=’, ’+’, ’-’, ’*’, ’/’, ’(’, ’)’, ’{’, ’}’, ’[’, ’,’, ’]’, ’.’, ’@’]

def t_STRING(t):
    r’\"(\\.|[^"])*\"|\’(\\.|[^"])*\’’
    t.value = t.value.strip(’"’).strip("’")
    return t

def t_ID(t):
    r’[a-zA-Z_][a-zA-Z0-9_]*’
    t.type = reserved.get(t.value, ’ID’) # Check for reserved words
    return t

def t_NUM(t):
    r’\d+|\d+\.\d+’
    try:
        t.value = int(t.value)
    except ValueError:
        print ’Integer value too large’, t.value
        t.value = 0
    return t

t_COMPOP = r’==|<=|>=|<|>|!=’
t_AUGASSIGN = r’\+=|-=|\*=|/=’
t_IN = r’:=’

t_ignore = " \t"

def t_NEWLINE(t):
    r’\n\s+’
    t.lexer.lineno += t.value.count(’\n’)
    return t

def t_error(t):
    print ’Illegal character "%s"’ % t.value[0]

# Build the lexer
lexer = lex.lex()
```

```

# Parsing rules
precedence = (
    (’left’,’+’,’-’),
    (’left’,’*’,’/’),
    (’left’, ’OR’),
    (’left’, ’AND’),
    (’left’, ’COMPOP’),
    (’left’, ’TO’),
    (’right’, ’NOT’),
    (’right’,’UMINUS’),
    (’right’, ’(’),
    (’left’, ’[’),
    (’left’, ’.’)
)

# Simple expressions

@register(’expr’)
def p_id(p):
    """id : ID"""
    p[0] = ast.Name(p[1], ast.Load())

def p_store_id(p):
    """store_id : ID"""
    p[0] = ast.Name(p[1], ast.Store())

def p_assign_id(p):
    """assign_id : assign_lst"""
    p[0] = ast.Tuple(p[1], ast.Store()) if len(p[1]) > 1 else p[1][0]

def p_assign_lst(p):
    """assign_lst : store_id ’,’ assign_lst
                  | store_id"""
    p = listify(p)

def p_store_property(p):
    """store_id : property"""
    p[1].ctx = ast.Store()
    p[0] = p[1]

def p_store_getitem(p):
    """store_id : getitem"""
    p[1].ctx = ast.Store()
    p[0] = p[1]

@register(’expr’)
def p_num(p):
    """num : NUM"""
    p[0] = ast.Num(p[1])

# Groupings

def p_expr_group(p):
    """expr : ’(’ expr ’)’"""
    p[0] = p[2]

# Strings

@register(’expr’)
def p_str(p):
    """str : STRING"""
    p[0] = ast.Str(p[1])

# Statements

def p_stmt_expr(p):
    """stmt : expr"""
    p[0] = ast.Expr(p[1])

def p_stmt_assignment(p):
    """stmt : assign_id ’=’ expr"""
    p[0] = ast.Assign([p[1]], p[3])

def p_stmt_aug_assignment(p):
    """stmt : store_id AUGASSIGN expr"""
    symbol_conversions = {
        ’+=’: ast.Add,
        ’-=’: ast.Sub,
        ’*=’: ast.Mult,
        ’/=’: ast.Div
    }
    p[0] = ast.AugAssign(p[1], symbol_conversions[p[2]](), p[3])

def p_stmt_return(p):
    """stmt : RETURN expr
            | RETURN"""
    if len(p) > 2:
        p[0] = ast.Return(p[2])
    else:
        p[0] = ast.Return(None)

def p_stmt_print(p):
    """stmt : PRINT expr"""
    p[0] = ast.Print(None, p[2] if isinstance(p[2], list) else [p[2]], True)

# Functions

@register(’stmt’)
def p_top_func(p):
    """topfunc : FUNC ’(’ params ’)’ ’{’ opt_newline body ’}’"""
    if p[3]:
        args = ast.arguments([ast.Name(’self’, ast.Param())], None, None, [])
    else:
        args = ast.arguments([], None, None, [])
    p[7] = [RewriteInjected([param[0].id for param in p[3]]).visit(node) for node in p[7]]
    p[0] = [ast.FunctionDef("top", args, p[7], [])]

@register(’stmt’)
def p_func(p):
    """func : FUNC ID ’(’ params ’)’ ’{’ opt_newline body ’}’"""
    if p[4]:
        arg_names, defaults = tuple([filter(lambda x: x is not None, item) for item in zip(*p[4])])
        args = ast.arguments(list(arg_names), None, None, list(defaults))
    else:
        args = ast.arguments([], None, None, [])
    p[0] = ast.FunctionDef(p[2], args, p[8], [])

@register(’expr’)
def p_funccall(p):
    """funccall : expr ’(’ opt_newline expr_list ’)’"""
    keywords = filter(lambda x: isinstance(x, ast.keyword), p[4])
    exprs = filter(lambda x: not isinstance(x, ast.keyword), p[4])
    p[0] = ast.Call(p[1], exprs, keywords, None, None)

@register(’expr’)
def p_lambda(p):
    """lambda : ’@’ ’(’ params ’)’ expr"""
    if p[3]:
        arg_names, defaults = tuple([filter(lambda x: x is not None, item) for item
in zip(*p[3])])
        args = ast.arguments(list(arg_names), None, None, list(defaults))
    else:
        args = ast.arguments([], None, None, [])
    p[0] = ast.Lambda(args, p[5])

def p_body(p):
    """body : stmtlst
            | empty"""
    if p[1]:
        p[0] = p[1]
    else:
        p[0] = [ast.Pass()]

p_opt_newline = trivial(’opt_newline’, [’NEWLINE’, ’empty’])

# Boolean logic

@register(’expr’)
def p_compare(p):
    """compare : expr COMPOP expr"""
    symbol_conversions = {
        ’==’: ast.Eq,
        ’!=’: ast.NotEq,
        ’<=’: ast.LtE,
        ’>=’: ast.GtE,
        ’<’: ast.Lt,
        ’>’: ast.Gt
    }

    p[0] = ast.Compare(p[1], [symbol_conversions[p[2]]()], [p[3]])

def p_bool_expr(p):
    """expr : expr AND expr
            | expr OR expr"""
    symbol_conversion = {
        ’and’: ast.And,
        ’or’: ast.Or
    }
    if isinstance(p[1], ast.BoolOp) and isinstance(p[1].op, symbol_conversion[p[2]])
:
        p[1].values.append(p[3])
        p[0] = p[1]
    else:
        p[0] = ast.BoolOp(symbol_conversion[p[2]](), [p[1], p[3]])

def p_expr_not(p):
    """expr : NOT expr %prec NOT"""
    p[0] = ast.UnaryOp(ast.Not(), p[2])

# Conditionals

@register(’stmt’)
def p_if(p):
    """if : IF expr ’{’ opt_newline body ’}’ opt_else"""
    p[0] = ast.If(p[2], p[5], p[7])

def p_opt_else(p):
    """opt_else : ELSE ’{’ opt_newline body ’}’
                | empty"""
    if len(p) > 2:
        p[0] = p[4]
    else:
        p[0] = []

def p_opt_elseif(p):
    """opt_else : ELSE expr ’{’ opt_newline body ’}’ opt_else"""
    p[0] = [ast.If(p[2], p[5], p[7])]

# Loops
@register(’stmt’)
def p_while(p):
    """while : WHILE expr ’{’ opt_newline body ’}’"""
    p[0] = ast.While(p[2], p[5], [])

@register(’stmt’)
def p_for(p):
    """for : FOR ID IN expr ’{’ opt_newline body ’}’"""
    p[0] = ast.For(ast.Name(p[2], ast.Store()), p[4], p[7], [])

@register(’expr’)
def p_range(p):
    """to : expr TO expr"""
    p[0] = ast.Call(ast.Name(’range’, ast.Load()), [p[1], p[3]], [], None, None)

# Lists

def p_params(p):
    """params : param ’,’ opt_newline params
              | param"""
    p = listify(p, list_pos=4)

def p_param(p):
    """param : ID
             | ID ’=’ expr
             | empty"""
    if p[1]:
        p[0] = (ast.Name(p[1], ast.Param()), None if len(p) < 3 else p[3])

def p_stmtlst(p):
    """stmtlst : stmt NEWLINE stmtlst
               | stmt opt_newline"""
    p = listify(p, size_check=3)

def p_in_params(p):
    """expr_list : opt_expr ’,’ opt_newline expr_list
                 | opt_expr"""
    p = listify(p, list_pos=4)

p_opt_expr = trivial(’opt_expr’, [’expr’, ’empty’])

def p_opt_expr_default(p):
    """opt_expr : ID ’=’ expr"""
    p[0] = ast.keyword(p[1], p[3])

@register(’expr’)
def p_list_braces(p):
    """list : ’[’ expr_list ’]’"""
    p[0] = ast.List(p[2], ast.Load())

# Property access

@register(’expr’)
def p_expr_property(p):
   """property : expr ’.’ ID"""

   p[0] = ast.Attribute(p[1], p[3], ast.Load())

@register(’expr’)
def p_expr_getitem(p):
    """getitem : expr ’[’ expr ’]’"""
    p[0] = ast.Subscript(p[1], ast.Index(p[3]), ast.Load())
```

```

# Arithmetic

def p_expr_binop(p):
    """expr : expr '+' expr
            | expr '-' expr
            | expr '*' expr
            | expr '/' expr"""
    if   p[2] == '+': p[0] = ast.BinOp(p[1], ast.Add(), p[3])  # p[1] + p[3]
    elif p[2] == '-': p[0] = ast.BinOp(p[1], ast.Sub(), p[3])  # p[1] - p[3]
    elif p[2] == '*': p[0] = ast.BinOp(p[1], ast.Mult(), p[3]) # p[1] * p[3]
    elif p[2] == '/': p[0] = ast.BinOp(p[1], ast.Div(), p[3])  # p[1] / p[3]

def p_expr_uminus(p):
    """expr : '-' expr %prec UMINUS"""
    if isinstance(p[2], ast.Num):
        p[2].n *= -1
        p[0] = p[2]
    else:
        p[0] = ast.UnaryOp(ast.USub(), p[2])

# Terminal registration

p_expr_reg = gen_function('expr')
p_stmt_reg = gen_function('stmt')

# Meta terminals

# Globals for communicating with p_error
# This is a code smell, but I don't think there's any easy way of
# communicating this otherwise
LINE_OFFSET = 1
COL_OFFSET = 1
FUNC_STR = ''

def p_error(p):
    print '[%d:%d] Syntax error at "%s"' % (p.lineno + LINE_OFFSET - 1, find_column(FUNC_STR, p) + COL_OFFSET - 2, p.value)

def p_empty(p):
    """empty :"""
    pass

test_parser = yacc.yacc(start='stmtlst')
parser = yacc.yacc(start='topfunc')

class BadParseException(Exception):
    def __init__(self, *args, **kwargs):
        super(self, BadParseException).__init__(*args, **kwargs)

def parse_string(s, debug=False, testing=False):
    """Parses a given string into a Python AST

    Args:
        s (String): The string to parse into an AST

    Named Args:
        debug (Bool): False -- A boolean representing whether to print debug info
        testing (Bool): False -- A boolean representing whether to use 'stmtlst' or 'topfunc' as the starting symbol

    Returns:
        ast.Module. The AST representation of the provided code string
    """
    if testing:
        body = test_parser.parse(s.strip(), debug=debug, lexer=lexer)
    else:
        body = parser.parse(s.strip(), debug=debug, lexer=lexer)
    return ast.Module(body)

def parse_function(func_str, name='top', debug=False, line_offset=1, col_offset=1):
    """Parses a string representing a Skit function into a first-class Python function

    Args:
        func_str (String): The string representing a Skit function to parse into a Python function

    Named Args:
        name (String): 'top' -- A string representing the name to give the function being parsed
        debug (Bool): False -- A boolean representing whether to print debug info
        line_offset (Int): 0 -- An int representing the line offset at which the function was found
        col_offset (Int): 0 -- An int representing the column offset at which the function was found

    Returns:
        Func. A first-class Python function that performs the actions of the Skit function provided
    """
    global LINE_OFFSET
    global COL_OFFSET
    global FUNC_STR
    LINE_OFFSET = line_offset
    COL_OFFSET = col_offset
    FUNC_STR = func_str

    func_ast = ast.fix_missing_locations(parse_string(func_str, debug=debug))

    exec(compile(func_ast, filename='<ast>', mode='exec'))
    locals()[name].__name__ = locals()[name].func_name = name
    return locals()[name]

env = locals()

def print_grammar():
    """Prints the grammar formed by the functions in this file
    """
    p_funcs = [func for name, func in env.items() if
               name.startswith('p_') and
               hasattr(func, '__call__') and
               name != 'p_error']
    grammar = defaultdict(list)
    for name, nonterminals in [func.__doc__.split(':') for func in p_funcs]:
        grammar[name.strip()].append(nonterminals)
    grammar = {key: [item for item in flatten(
        [[docstr.strip() for docstr in item.split('|')] for item in value]
    )] for key, value in grammar.iteritems()}

    for name, nonterminals in grammar.iteritems():
        print gen_grammar(name, sorted(nonterminals), indent=0) + '\n'

if __name__ == '__main__':
    while 1:
        try:
            s = raw_input('>')
        except EOFError:
            break
        if not s: continue
        print ast.dump(parse_string(s))
```

```
import unittest
import ast

#TODO fix relative import
from ..parser import parse_string, parse_function
from ..oracle import ORACLE

class ParsingASTTests(unittest.TestCase):
    def assertSameParse(self, python, skit):
        self.assertEqual(
            ast.dump(ast.parse(python)),
            ast.dump(parse_string(skit, testing=True))
        )

    def test_id(self):
        self.assertSameParse("test", "test")

    def test_num(self):
        self.assertSameParse("1", "1")

    def test_group(self):
        self.assertSameParse("(1 + 2)", "(1 + 2)")

    def test_string_single_quotes(self):
        self.assertSameParse("’test’", "’test’")

    def test_string_double_quotes(self):
        self.assertSameParse(’"test"’, ’"test"’)

    def test_stmt_assignment(self):
        self.assertSameParse("test = 1", "test = 1")

    def test_multi_stmt_assignment(self):
        self.assertSameParse("a, b = tpl", "a, b = tpl")

    def test_stmt_assign_property(self):
        self.assertSameParse("a.b.c = 1", "a.b.c = 1")

    def test_stmt_assign_getitem(self):
        self.assertSameParse("a[’b’][’c’] = 1", ’a["b"]["c"] = 1’)

    def test_stmt_aug_assign_add(self):
        self.assertSameParse("test += 1", "test += 1")

    def test_stmt_aug_assign_sub(self):
        self.assertSameParse("test -= 1", "test -= 1")

    def test_stmt_aug_assign_mult(self):
        self.assertSameParse("test *= 1", "test *= 1")

    def test_stmt_aug_assign_div(self):
        self.assertSameParse("test /= 1", "test /= 1")

    def test_stmt_return(self):
        self.assertSameParse("return", "return")

    def test_stmt_return_value(self):
        self.assertSameParse("return 1", "return 1")

    def test_stmt_print(self):
        self.assertSameParse("print 1", "print 1")

    def test_func(self):
        self.assertSameParse("def test(): pass",
                             "func test() { }")

    def test_func_param(self):
        self.assertSameParse("def test(one): pass",
                             "func test(one) { }")

    def test_func_default_param(self):
        self.assertSameParse("def test(one=1): pass",
                             "func test(one=1) { }")

    def test_func_body(self):
        self.assertSameParse("def test(): return",
                             "func test() { return }")

    def test_lambda(self):
        self.assertSameParse("lambda x: x",
                             "@(x) x")

    def test_funccall(self):
        self.assertSameParse("test()", "test()")

    def test_funccall_param(self):
        self.assertSameParse("test(1)", "test(1)")

    def test_funccall_params(self):
        self.assertSameParse("test(1,2)", "test(1,2)")

    def test_funccall_keyword_param(self):
        self.assertSameParse("test(one=1)", "test(one=1)")

    def test_funccall_keyword_params(self):
        self.assertSameParse("test(one=1,two=2)", "test(one=1,two=2)")

    def test_cond_eq(self):
        self.assertSameParse("1 == 1", "1 == 1")

    def test_cond_neq(self):
        self.assertSameParse("1 != 2", "1 != 2")

    def test_cond_lte(self):
        self.assertSameParse("1 <= 2", "1 <= 2")

    def test_cond_gte(self):
        self.assertSameParse("2 >= 1", "2 >= 1")

    def test_cond_lt(self):
        self.assertSameParse("1 < 2", "1 < 2")

    def test_cond_gt(self):
        self.assertSameParse("2 > 1", "2 > 1")

    def test_true(self):
        self.assertSameParse("True", "True")

    def test_false(self):
        self.assertSameParse("False", "False")

    def test_and(self):
        self.assertSameParse("True and False", "True and False")

    def test_and_chain(self):
        self.assertSameParse("True and False and True", "True and False and True")

    def test_or(self):
        self.assertSameParse("True or False", "True or False")

    def test_or_chain(self):
        self.assertSameParse("True or False or True", "True or False or True")

    def test_and_or(self):
        self.assertSameParse("True and False or True", "True and False or True")

    def test_or_and(self):
        self.assertSameParse("True or False and True", "True or False and True")

    def test_and_or_chain(self):
        self.assertSameParse("True and False or True or False", "True and False or True or False")

    def test_or_and_chain(self):
        self.assertSameParse("True or False and True and False", "True or False and True and False")

    def test_or_and_or_chain(self):
        self.assertSameParse("True or False and True or False", "True or False and True or False")

    def test_and_or_and_chain(self):
        self.assertSameParse("True and False or True and False", "True and False or True and False")

    def test_and_compop(self):
        self.assertSameParse("1 >= 2 and 3 <= 4", "1 >= 2 and 3 <= 4")

    def test_or_compop(self):
        self.assertSameParse("1 >= 2 or 3 <= 4", "1 >= 2 or 3 <= 4")

    def test_not(self):
        self.assertSameParse("not False", "not False")

    def test_if(self):
        self.assertSameParse("if 1: pass",
                             "if 1 { }")

    def test_if_cond(self):
        self.assertSameParse("if 1 == 1: pass",
                             "if 1 == 1 { }")

    def test_if_body(self):
        self.assertSameParse("if 1: print 1",
                             "if 1 { print 1 }")

    def test_if_else(self):
        self.assertSameParse("if 1:\n  pass\nelse:\n  pass",
                             "if 1 { } else { }")

    def test_if_else_body(self):
        self.assertSameParse("if 1:\n  print 1\nelse:\n  print False",
                             "if 1 { print 1 } else { print False }")

    def test_if_elseif(self):
        self.assertSameParse("if 1:\n  pass\nelif 2:\n  pass",
                             "if 1 { } else 2 { }")

    def test_if_elseif_chain(self):
        self.assertSameParse("if 1:\n  pass\nelif 2:\n  pass\nelif 3:\n  pass",
                             "if 1 { } else 2 { } else 3 { }")

    def test_if_elseif_else(self):
        self.assertSameParse("if 1:\n  pass\nelif 2:\n  pass\nelse:\n  pass",
                             "if 1 { } else 2 { } else { }")

    def test_if_elseif_chain_else(self):
        self.assertSameParse("if 1:\n  pass\nelif 2:\n  pass\nelif 3:\n  pass\nelse:\n  pass",
                             "if 1 { } else 2 { } else 3 { } else { }")

    #TODO Add ternary operator
    #def test_ternary(self):
    #    self.assertSameParse("1 if True else 2",
    #                         "True ? 1 : 2")

    def test_while(self):
        self.assertSameParse("while 1: pass",
                             "while 1 { }")

    def test_while_body(self):
        self.assertSameParse("while 1: print 1",
                             "while 1 { print 1 }")

    def test_for(self):
        self.assertSameParse("for i in range(1,2): pass",
                             "for i := range(1,2) {}")

    def test_for_body(self):
        self.assertSameParse("for i in range(1,2): print i",
                             "for i := range(1,2) { print i }")

    def test_list_decl(self):
        self.assertSameParse("[1,2,3]", "[1,2,3]")

    def test_property(self):
        self.assertSameParse("test.test", "test.test")

    def test_getitem(self):
        self.assertSameParse("test[test]", "test[test]")

    def test_binop_plus(self):
        self.assertSameParse("1 + 1", "1 + 1")

    def test_binop_minus(self):
        self.assertSameParse("1 - 1", "1 - 1")

    def test_binop_times(self):
        self.assertSameParse("1 * 1", "1 * 1")

    def test_binop_div(self):
        self.assertSameParse("1 / 1", "1 / 1")

    def test_uminus(self):
        self.assertSameParse("-1", "-1")

class ParsingBehaviorTests(unittest.TestCase):
    def assertSameParse(self, skit1, skit2):
        self.assertEqual(
            ast.dump(parse_string(skit1)),
            ast.dump(parse_string(skit2))
        )

    def compileFunc(self, func):
        return parse_function(func)

    def assertResult(self, func, result, eq=True):
        if eq:
            self.assertEqual(result, func({}))
        else:
            self.assertNotEqual(result, func({}))

    def test_group_same_as_regular(self):
        self.assertSameParse("1 + 2", "(1 + 2)")

    def test_single_double_qoutes(self):
        self.assertSameParse("’test’", ’"test"’)

    def test_range(self):
        self.assertSameParse("range(1,2)", "1 to 2")

    def test_top_func(self):
        test = []
        ORACLE.set(’test’, test)

        func = self.compileFunc("func(test) { return test }")
        test.append(1)

        self.assertResult(func, test)

        test.pop()
        self.assertResult(func, test)

        test = [1,2,3]
        self.assertResult(func, test, eq=False)

        ORACLE.set(’test’, test)
        self.assertResult(func, test)
```

```python
from itertools import imap, chain
from collections import Sequence

def listlike(obj):
    """Checks if the object is like a sequential container

    Args:
        obj (Object): The object to check

    Returns:
        Bool. True if the object is listlike, False if it’s a string

    """
    return isinstance(obj, Sequence) and not isinstance(obj, basestring)


def one_or_many(value):
    """Ensures the value can be used like a list

    Args:
        value (Any): The value to check

    Returns:
        Any. The value if it’s listlike, or the value wrapped in a tuple if it isn’t

    """
    return value if listlike(value) else (value,)


def flatten(values):
    """Iterate over objects like a flat list

    Args:
        values (List): A list of objects to flatten

    Returns:
        List. A list containing the nested objects in values
    """
    return chain.from_iterable(imap(one_or_many, values))

def find_column(input, token=None, lexpos=None):
    """Finds the column of a token given the input it’s in

    Args:
        input (String) - The input being parsed
        token (Token) - The token being located

    Returns:
        The column the token being located is in
    """
    lexpos = lexpos or token.lexpos
    last_cr = input.rfind(’\n’,0,lexpos)
    if last_cr < 0:
        last_cr = 0
    column = (lexpos - last_cr) + 1
    return column
```

```makefile
# makefile

.PHONY: clean
clean:
	find . -name "*.pyc" -exec rm -rf {} \;
```