

```

1: import ply.lex as lex
2: import ply.yacc as yacc
3: import sys
4: import os
5: sys.path.append('.')
6: from imperative_parser.parser import parse_function
7: from imperative_parser.utils import find_column
8:
9: tokens = (
10:     'COLON',
11:     'LCURLY',
12:     'RCURLY',
13:     'LBRACKET',
14:     'RBRACKET',
15:     'COMMA',
16:     'DOT',
17:     'WILD',
18:     'PLUS',
19:     'ID',
20:     'EXTENSION',
21:     'STR',
22:     'FUNC',
23:     'NUM'
24: )
25:
26: reserved = {
27:     'uniform': 'UNIFORM',
28:     'None': 'NONE'
29: }
30:
31: tokens += tuple(reserved.values())
32: t_COLON = r':'
33: t_LCURLY = r'\{'
34: t_RCURLY = r'\}'
35: t_LBRACKET = r'\['
36: t_RBRACKET = r'\]'
37: t_COMMA = r','
38: t_DOT = r'\.'
39: t_WILD = r'\*'
40: t_PLUS = r'\+'
41: t_STR = r'".*"'
42: t_ignore = ' \t'
43:
44: def t_FUNC(t):
45:     r'func[^\{]*{'
46:     func = t.value
47:     bracks = 1
48:     pos = t.lexer.lexpos
49:     pos2 = t.lexpos
50:     lexdata = t.lexer.lexdata[t.lexer.lexpos:]
51:     for c in lexdata:
52:         t.lexer.lexpos += 1
53:         func += c
54:         if c == '{':
55:             bracks += 1
56:         elif c == '}':
57:             bracks -= 1
58:         elif c == '\n':
59:             t.lexer.lineno += 1
60:         if not bracks:
61:             break
62:     t.value = func
63:     return t
64:
65: def t_ID(t):
66:     r'[A-Za-z][A-Za-z-]*'

```

```

67:     t.type = reserved.get(t.value, 'ID')
68:     return t
69:
70: def t_EXTENSION(t):
71:     r'@extend'
72:     return t
73:
74: def t_NUM(t):
75:     r'\d+'
76:     t.value = int(t.value)
77:     return t
78:
79: def t_newline(t):
80:     r'\n+'
81:     t.lexer.lineno += len(t.value)
82:
83: def t_error(t):
84:     print "Illegal character '%s'" % t.value[0]
85:
86: lexer = lex.lex()
87:
88: # Error Handling
89: SUCCEEDED = True
90: PARSED_STRING = ""
91:
92: def p_property_value(p):
93:     'property : ID COLON value'
94:     p[0] = {p[1]: p[3]}
95:
96: def p_property_extension(p):
97:     'property : EXTENSION COLON value'
98:     p[0] = {p[1]: p[3]}
99:
100: def p_value_structure(p):
101:     'value : structure'
102:     p[0] = p[1]
103:
104: def p_value_list(p):
105:     'value : LBRACKET list RBRACKET'
106:     p[0] = p[2]
107:
108: def p_value_dots(p):
109:     'value : dots'
110:     p[0] = p[1]
111:
112: def p_value_num(p):
113:     'value : NUM'
114:     p[0] = p[1]
115:
116: def p_value_str(p):
117:     'value : STR'
118:     p[0] = p[1].strip('\''')
119:
120: def p_value_uniform(p):
121:     'value : UNIFORM'
122:     p[0] = 'uniform'
123:
124: def p_value_none(p):
125:     'value : NONE'
126:     p[0] = None
127:
128: def p_value_func(p):
129:     'value : FUNC'
130:     global SUCCEEDED
131:     try:
132:         p[0] = parse_function(p[1], line_offset=p.lineno(1), col_offset=find_column(

```

```
PARSED_STRING, lexpos=p.lexpos(1)))
133:     except:
134:         SUCCEEDED = False
135:         p[0] = p[1]
136:
137: def p_structure_properties(p):
138:     'structure : LCURLY properties RCURLY'
139:     p[0] = p[2]
140:
141: def p_list_comma(p):
142:     'list : value COMMA list'
143:     p[1] = [p[1]]
144:     p[1].extend(p[3])
145:     p[0] = p[1]
146:
147: def p_list_value(p):
148:     'list : value'
149:     p[0] = [p[1]]
150:
151: def p_dots_dot(p):
152:     'dots : ID DOT dots'
153:     p[0] = p[1] + '.' + p[3]
154:
155: def p_dots_plus(p):
156:     'dots : ID PLUS NUM'
157:     p[0] = p[1] + ' ' + ' ' + str(p[3])
158:
159: def p_dots_id(p):
160:     'dots : ID'
161:     p[0] = p[1]
162:
163: def p_dots_wild(p):
164:     'dots : WILD'
165:     p[0] = '*'
166:
167: def p_properties_comma(p):
168:     'properties : property COMMA properties'
169:     p[3].update(p[1])
170:     p[0] = p[3]
171:
172: def p_properties_property(p):
173:     'properties : property'
174:     p[0] = p[1]
175:
176: def p_error(p):
177:     print p
178:     print "Syntax error in input!"
179:
180: parser = yacc.yacc()
181:
182: def parse(s):
183:     global SUCCEEDED
184:     global PARSED_STRING
185:     PARSED_STRING = s
186:     return parser.parse(s, lexer=lexer), SUCCEEDED
```

```

1:
2: # parsetab.py
3: # This file is automatically generated. Do not edit.
4: _tabversion = '3.2'
5:
6: _lr_method = 'LALR'
7:
8: _lr_signature = '\x04\x0d\x11K\x81t\x9c\x08\x8f8\xdb\x0f_\xf3'
9:
10: _lr_action_items = {'PLUS':([16,],[23,]),'NONE':([4,5,11,26,],[7,7,7,]),'FUNC':([4,5,11,26,],[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,]),'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,]),}
11:
12: _lr_action = { }
13: for _k, _v in _lr_action_items.items():
14:     for _x,_y in zip(_v[0],_v[1]):
15:         if not _x in _lr_action: _lr_action[_x] = { }
16:         _lr_action[_x][_k] = _y
17: del _lr_action_items
18:
19: _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,]),}
20:
21: _lr_goto = { }
22: for _k, _v in _lr_goto_items.items():
23:     for _x,_y in zip(_v[0],_v[1]):
24:         if not _x in _lr_goto: _lr_goto[_x] = { }
25:         _lr_goto[_x][_k] = _y
26: del _lr_goto_items
27: _lr_productions = [
28:     ("S" -> property", "S", 1, None, None, None),
29:     ('property -> ID COLON value', 'property', 3, 'p_property_value', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 93),
30:     ('property -> EXTENSION COLON value', 'property', 3, 'p_property_extension', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 97),
31:     ('value -> structure', 'value', 1, 'p_value_structure', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 101),
32:     ('value -> LBRACKET list RBRACKET', 'value', 3, 'p_value_list', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 105),
33:     ('value -> dots', 'value', 1, 'p_value_dots', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 109),
34:     ('value -> NUM', 'value', 1, 'p_value_num', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 113),
35:     ('value -> STR', 'value', 1, 'p_value_str', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 117),
36:     ('value -> UNIFORM', 'value', 1, 'p_value_uniform', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 121),
37:     ('value -> NONE', 'value', 1, 'p_value_none', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 125),
38:     ('value -> FUNC', 'value', 1, 'p_value_func', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 129),
39:     ('structure -> LCURLY properties RCURLY', 'structure', 3, 'p_structure_properties', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 138),
40:     ('list -> value COMMA list', 'list', 3, 'p_list_comma', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 142),
41:     ('list -> value', 'list', 1, 'p_list_value', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 148),

```

```

42:     ('dots -> ID DOT dots', 'dots', 3, 'p_dots_dot', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 152),
43:     ('dots -> ID PLUS NUM', 'dots', 3, 'p_dots_plus', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 156),
44:     ('dots -> ID', 'dots', 1, 'p_dots_id', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 160),
45:     ('dots -> WILD', 'dots', 1, 'p_dots_wild', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 164),
46:     ('properties -> property COMMA properties', 'properties', 3, 'p_properties_comma', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 168),
47:     ('properties -> property', 'properties', 1, 'p_properties_property', '/Users/mdzhang/Projects/pltcatan/config_parser/config.py', 173),
48: ]

```



```

1: #!/usr/bin/env python
2: import config
3: import argparse
4: import dill as pickle
5: import os
6: import shutil
7: import sys
8: sys.path.append('.')
9: from engine.src.game import Game
10: from engine.src.config.config import Config
11:
12: properties = {}
13:
14: def undot(property):
15:     """
16:     Get the value of a dot-notated.property from the properties dict
17:     """
18:     extended = properties
19:     extension = property.split('.')
20:     extension.reverse()
21:     while extension:
22:         extended = extended.get(extension.pop(), properties)
23:         if extended is properties:
24:             return extended
25:     if isinstance(extended, dict) or isinstance(extended, list):
26:         return extended.copy()
27:     else:
28:         return extended
29:
30: def extend_verbose(skit, property, value, extension):
31:     """
32:     Extend properties using the verbose syntax where every extension must use an
33:     @extend explicitly
34:     """
35:     skit[property] = undot(extension)
36:     for extended_property, extended_value in value.iteritems():
37:         if extended_property != '@extend':
38:             if isinstance(extended_value, str) and '+' in extended_value:
39:                 extension, addition = extended_value.split('+')
40:                 extended = undot(extension.strip())
41:                 extended_value = extended + int(addition)
42:                 skit[property][extended_property] = extended_value
43:
44: def extend_clean(skit, property, value, extension):
45:     """
46:     Extend properties using the cleaner syntax where one mention of @extend and
47:     explicit-overwrite-only set to true cascades the extension gracefully
48:     """
49:     explicit = extension['explicit-overwrite-only']
50:     extension = extension['value']
51:     extend_verbose(skit, property, value, extension)
52:     if explicit:
53:         for extended_property, extended_value in value.iteritems():
54:             if isinstance(extended_value, dict) and extended_property != \
55:                 '@extend':
56:                 if needs_extending(extended_value):
57:                     skit[property][extended_property]['@extend'] = \
58:                         make_extend(extension, extended_property, explicit)
59:     return extension
60:
61: def needs_extending(skit):
62:     """
63:     Checks to see if a structure needs to be extended
64:     """
65:     children_structures = False
66:     if isinstance(skit, dict):

```

```

67:         return True
68:     for property, value in skit.iteritems():
69:         if isinstance(value, dict):
70:             children_structures = True
71:     return children_structures
72:
73: def make_extend(extension, extended_property, explicit):
74:     """
75:     Coerce the structure to look like a verbose extension
76:     """
77:     return {'value': '%s.%s' % (extension, extended_property),
78:             'explicit-overwrite-only': explicit}
79:
80: def replace(value):
81:     """
82:     Replace an import alias with its actual value
83:     """
84:     if '+' in value:
85:         terms = value.split('+')
86:         sum = 0
87:         for term in terms:
88:             term = term.strip()
89:             if term.isdigit():
90:                 replacement = float(term)
91:             else:
92:                 replacement = undot(term.strip())
93:                 if replacement is properties:
94:                     sum = None
95:                     break
96:             sum += float(replacement)
97:         if sum is None:
98:             replacement = value
99:         else:
100:             replacement = sum
101:     else:
102:         replacement = undot(value.strip())
103:     if replacement is properties:
104:         return value
105:     else:
106:         return replacement
107:
108: def extend(skit, parent=None):
109:     """
110:     Replace all extended properties with the contents of the actual value
111:     denoted by the dot-notated property name and set any additional properties
112:     """
113:     for property, value in skit.iteritems():
114:         if isinstance(value, str):
115:             replacement = replace(value)
116:             if isinstance(replacement, dict):
117:                 replacement = replacement.get(property, replacement)
118:                 skit[property] = replacement
119:             if isinstance(value, dict):
120:                 extension = value.get('@extend')
121:                 if extension:
122:                     if isinstance(extension, str):
123:                         extend_verbose(skit, property, value, extension)
124:                     extension = None
125:                 else:
126:                     extension = extend_clean(skit, property, value, extension)
127:                 extend(skit[property])
128:
129: def imports(full_file, file):
130:     """
131:     Compiles every skit structure that is imported in addition to
132:     the top-level structure

```

```

133:     '''
134:     imports = file.split('\n')
135:     line_no = 0
136:     chars_read = 0
137:     for line in imports:
138:         line_length = len(line)
139:         if line:
140:             line = line.split()
141:             if line[0] == '@import':
142:                 if len(line) < 4:
143:                     print 'Error: Invalid @import on line', line_no
144:                     return None
145:                 if line[1][-1] == '//':
146:                     if line[1][0] == '.':
147:                         properties[line[3]], success = compile(full_file + line[1] +
148:
149:                             '.__value__.skit', as_name=line[3])
150:                     elif line[1][0] == '/':
151:                         properties[line[3]], success = compile(line[1] + \
152:                             '.__value__.skit', as_name=line[3])
153:                     else:
154:                         if line[1][0] == '.':
155:                             properties[line[3]], success = compile(full_file + line[1] +
156:
157:                                 '.__skit')
158:                         elif line[1][0] == '/':
159:                             properties[line[3]], success = compile(line[1] + '.__skit')
160:                         else:
161:                             break
162:                         line_no += 1
163:                         chars_read += line_length
164:                     if chars_read > 0:
165:                         chars_read += 1
166:                     return file[chars_read:]
167: def compile(file, clean=False, as_name=None):
168:     '''
169:     Cleans tmp/ directory and reinitializes with compiled skit code
170:     '''
171:     full_file = os.path.dirname(file) + '/'
172:     base_file = os.path.basename(file)
173:     compile_file = 'tmp/' + base_file
174:     if clean:
175:         shutil.rmtree('tmp/', True)
176:         compile('default.skit')
177:         file = open(file, 'r').read()
178:         file = imports(full_file, file)
179:         skit, succeeded = config.parse(file)
180:         main_property = os.path.splitext(base_file)[0]
181:         extend(skit)
182:         if as_name:
183:             properties[as_name] = skit
184:             main_property = as_name
185:         else:
186:             properties[main_property] = skit.get(main_property)
187:         if not os.path.isdir('tmp/'):
188:             os.makedirs('tmp/')
189:         pickle.dump(skit, open(compile_file, 'wb'))
190:         return skit, succeeded
191:
192: def run(file):
193:     '''
194:     Runs skit game
195:     Recompiles skit code only if code has been changed
196:     '''

```

```

197: _, success = compile('default.skit')
198: if success:
199:     base_file = os.path.basename(file)
200:     compile_file = 'tmp/' + base_file
201:     skit = None
202:     if not os.path.isfile(compile_file) or \
203:         os.path.getmtime(file) > os.path.getmtime(compile_file):
204:         skit = compile(file)[0]
205:     else:
206:         skit = pickle.load(open(compile_file, 'rb'))
207:     main_property = os.path.splitext(base_file)[0]
208:     properties[main_property] = skit.get(main_property)
209:     Config.config = properties[main_property]
210:     Config.init()
211:     game = Game()
212:     skit = skit.get(os.path.splitext(base_file)[0], None)
213:     # TODO: restore after engine syncs config dict format
214:     # if skit.get('game', None):
215:     game.start()
216: else:
217:     print "Build failed, check the log for errors"
218:     sys.exit(1)
219:
220: if __name__ == '__main__':
221:     arg_parser = argparse.ArgumentParser(description='Skit compiler')
222:     arg_parser.add_argument('file', help='Skit file')
223:     arg_parser.add_argument('-c', '--compile', action='store_true',
224:                             help='Only run compile steps')
225:     args = arg_parser.parse_args()
226:     if args.compile:
227:         compile(args.file, True)
228:     else:
229:         run(args.file)

```

```

1: #!/usr/bin/env python
2: import sys
3: import config
4: import skit
5:
6: passed_all = True
7:
8: def dummy():
9:     return 0
10:
11: recognized_types = [type(''), type(0), type(dict()), type(list()), type(None),\
12:                     type(dummy)]
13: function_names = ['play-card', 'draw-card']
14: string_names = ['name', 'description', 'position-type']
15: int_names = ['points-to-win', 'player-count', 'radius', 'tile-count', 'count',\
16:              'point-value', 'base-yield']
17: structure_names = ['game', 'board', 'card', 'development', 'structure',\
18:                   'player-built']
19:
20: def type_per_name(skit, property, value):
21:     can_be_none = False
22:     global passed_all
23:     if property in function_names:
24:         if type(value) != type(dummy):
25:             print 'Error: property %s does not contain a function' % property
26:             print 'Actual type: %s', type(value)
27:             passed_all = False
28:     elif property in string_names:
29:         if type(value) != type(''):
30:             print 'Error: property %s does not contain a string' % property
31:             print 'Actual type: %s', type(value)
32:             passed_all = False
33:     elif property in int_names:
34:         if type(value) != type(0):
35:             print 'Error: property %s does not contain an integer' % property
36:             print 'Actual type: %s', type(value)
37:             passed_all = False
38:     elif property in structure_names:
39:         if type(value) != type(dict()):
40:             print 'Error: property %s does not contain a dict' % property
41:             print 'Actual type: %s', type(value)
42:             passed_all = False
43:
44: def test_types(skit):
45:     if type(skit) not in recognized_types:
46:         print 'Error: %s has unrecognized type', (skit, type(skit))
47:     if isinstance(skit, dict):
48:         for property, value in skit.iteritems():
49:             if property == 'default' and not skit[property].get('game', None):
50:                 continue
51:             else:
52:                 type_per_name(skit, property, value)
53:                 test_types(value)
54:
55: if __name__ == '__main__':
56:     game = config.parser.parse(open('default.skit', 'r').read())
57:     default = skit.compile('default.skit')
58:     test_types(game)
59:     test_dict = {'test': {'game': {'points-to-win': 5 } } }
60:     test_skit = 'test: { game: { points-to-win: 5 } }'
61:     compiled_skit = config.parser.parse(test_skit)
62:     if test_dict != compiled_skit:
63:         print 'Error: Static test dict does not match compiled test.skit'
64:         print 'Static test dict: %s', test_dict
65:         print 'Compiled test.skit: %s', compiled_skit
66:         passed_all = False

```

```

67:     test_dict['test']['game']['points-to-win'] = 10
68:     if test_dict == compiled_skit:
69:         print 'Error: Static test dict matches compiled test.skit with lower \
70: points to win'
71:         print 'Static test dict: %s', test_dict
72:         print 'Compiled test.skit: %s', compiled_skit
73:         passed_all = False
74:     test_skit = 'test: { game: { points-to-win: default.game.points-to-win } }'
75:     compiled_skit = config.parser.parse(test_skit)
76:     skit.extend(compiled_skit)
77:     if test_dict != compiled_skit:
78:         print 'Error: Static test dict does not match compiled test.skit\'s \
79: points-to-win'
80:         print 'Static test dict: %s', test_dict
81:         print 'Compiled test.skit: %s', compiled_skit
82:         passed_all = False
83:     test_skit = 'test: { game: default.game }'
84:     first_compile = config.parser.parse(test_skit)
85:     second_compile = config.parser.parse(test_skit)
86:     if first_compile != second_compile:
87:         print 'Error: Equivalent skit structures do not match when compiled'
88:         print 'Static test dict: %s', test_dict
89:         print 'Compiled test.skit: %s', compiled_skit
90:         passed_all = False
91:     skit.extend(first_compile)
92:     skit.extend(second_compile)
93:     if first_compile != second_compile:
94:         print 'Error: Equivalent skit structures do not match when extended'
95:         print 'Static test dict: %s', test_dict
96:         print 'Compiled test.skit: %s', compiled_skit
97:         passed_all = False
98:     first_skit = 'skit: { one: { a: 5, b: 6, c: 4 }, two: { b: 6, a: 5, c: 4 } }'
99:     second_skit = 'skit: { two: { b: 6, a: 5, c: 4 }, one: { a: 5, b: 6, c: 4 } }'
100:    first_compile = config.parser.parse(first_skit)
101:    second_compile = config.parser.parse(second_skit)
102:    if first_compile != second_compile:
103:        print 'Error: Semantically skit structures do not match when extended'
104:        print 'Static test dict: %s', test_dict
105:        print 'Compiled test.skit: %s', compiled_skit
106:        passed_all = False
107:    if passed_all:
108:        print 'Passed every test!'

```



```
1: # makefile
2:
3: start: start.py
4:         python start.py
5:
6: debug:
7:         # pdb.set_trace()
8:         python -m pdb start.py
9:
10: .PHONY: clean
11: clean:
12:         find . -name "*.pyc" -exec rm -rf {} \;
```


`./engine/src/board/board.py`

Mon Mar 30 15:45:14 2015

1

```
1: # -*- coding: utf-8 -*-  
2:  
3:  
4: class Board(object):  
5:     pass
```



```

1: # -*- coding: utf-8 -*-
2: import random
3: import pdb
4:
5: from engine.src.lib.utils import Utils
6: from engine.src.board.hex_board import HexBoard
7: from engine.src.tile.game_tile import GameTile
8: from engine.src.resource_type import ResourceType
9: from engine.src.position_type import PositionType
10: from engine.src.calamity.calamity import Calamity
11: from engine.src.calamity.calamity import CalamityTilePlacementEffect
12: from engine.src.calamity.robber import Robber
13: from engine.src.trading.bank import Bank
14: from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping
15: from engine.src.direction.edge_direction import EdgeDirection
16: from engine.src.direction.vertex_direction import VertexDirection
17: from engine.src.exceptions import *
18: from engine.src.structure.structure import Structure
19:
20:
21: class GameBoard(HexBoard):
22:     """A Settlers of Catan playing board.
23:
24:     Attributes:
25:         radius (int): See HexBoard.
26:
27:         tiles (dict): See HexBoard.
28:
29:         tile_cls (class): See HexBoard.
30:
31:         bank (Bank): Bank of resources the board will interact with.
32:
33:     Args:
34:         radius (int): See HexBoard.
35:     """
36:
37:     def __init__(self, radius):
38:
39:         super(GameBoard, self).__init__(radius, GameTile)
40:
41:         # We have tiles, but they currently have no value and are all FALLOW.
42:         # Here we assign resource types and chit values.
43:         self.assign_tile_resources()
44:         self.assign_tile_chit_values()
45:         self.assign_tile_harbors()
46:
47:         self.bank = Bank(len(list(self.iter_tiles())))
48:
49:     def assign_tile_resources(self, assignment_func=None):
50:         """Assign resource types to this board's tiles.
51:
52:     Args:
53:         assignment_func (func): Resources will assigned according to this
54:             function. If not provided, will default to
55:             self._default_assign_tile_resources()
56:
57:     Returns:
58:         None.
59:     """
60:
61:     if assignment_func is None:
62:         self._default_assign_tile_resources()
63:     else:
64:         assignment_func()
65:
66:     def _default_assign_tile_resources(self):

```

```

67:
68:
69:     Specifically, assigns one ResourceType.FALLOW tile, then splits the
70:     resource types of the remaining tiles evenly.
71:
72:     Returns:
73:         None.
74:
75:     TODO: Defaults to only one FALLOW tile regardless of board size.
76:         Perhaps should make fallow tile count relative to board size.
77:     """
78:
79:     # Get a randomized list of the tiles of this board.
80:     tiles = list(self.iter_tiles())
81:     random.shuffle(tiles)
82:
83:     resource_type_count = len(ResourceType.get_arable_types())
84:
85:     # We'll allocate one fallow tile so divide arable resources among
86:     # total number of tiles - 1.
87:     per_resource_count = (len(tiles) - 1) / float(resource_type_count)
88:
89:     # Say that we find that we need to allocate 3.6 tiles per resource.
90:     # Clearly we can only allocate whole number tiles. So we take the
91:     # difference between what we calculated and its floor (e.g. .6),
92:     # and multiply it by the number of tiles to get the number of
93:     # leftover tiles that need to be assigned.
94:     leftover_count = int((per_resource_count - int(per_resource_count)) *
95:                          resource_type_count)
96:
97:     per_resource_count = int(per_resource_count)
98:
99:     # Get a list containing resource_type_count occurrences of each
100:     # resource_type.
101:     resources = Utils.flatten(map(
102:         lambda resource: [resource] * per_resource_count,
103:         ResourceType.get_arable_types()
104:     ))
105:
106:     # We then allocate leftover tiles according to some priority. In a
107:     # base Settlers of Catan game, this priority manifests as having only
108:     # 3 brick and ore tiles, by 4 lumber, wool, and wheat tiles.
109:     while leftover_count:
110:         resources.append(
111:             ResourceType.get_priority_arable_types()[leftover_count - 1])
112:         leftover_count -= 1
113:
114:     # Add a single occurrence of ResourceType.FALLOW.
115:     resources.append(ResourceType.FALLOW)
116:
117:     # Assign the resource types to the shuffled tiles.
118:     for tile, resource_type in zip(tiles, resources):
119:         tile.resource_type = resource_type
120:
121: def _randomly_assign_tile_resources(self):
122:     """Randomly assign resource types to this board's tiles.
123:
124:     Note that this randomly draws from all ResourceType's, i.e. including
125:     ResourceType.FALLOW.
126:
127:     Returns:
128:         None.
129:     """
130:
131:     for tile in self.iter_tiles():
132:         tile.resource_type = ResourceType.random()

```

```

133:
134:     def assign_tile_chit_values(self, assignment_func=None):
135:         """Assign chit values to this board's tiles.
136:
137:         Args:
138:             assignment_func (func): Chit values will assigned according to this
139:             function. If not provided, will default to
140:             self._default_assign_tile_chit_values()
141:
142:         Returns:
143:             None.
144:         """
145:
146:         if assignment_func is None:
147:             self._default_assign_tile_chit_values()
148:         else:
149:             assignment_func()
150:
151:     def _randomly_assign_tile_chit_values(self, start=2, end=12,
152:                                         exclude=Calamity.DEFAULT_ROLL_VALUES):
153:         """Randomly assign chit values to this board's tiles.
154:
155:         Args:
156:             start (int): The set of possible chit values from which values to
157:             assign will be randomly drawn is defined by the range defined by
158:             start and end.
159:
160:             end (int): See above.
161:
162:             exclude (list): A list of values that lie in the range given by
163:             start and end that should not be included in the set of possible
164:             chit values.
165:
166:         Returns:
167:             None
168:         """
169:
170:         chit_values = frozenset(range(start, end + 1)).intersection(exclude)
171:
172:         for tile in self.iter_tiles():
173:             tile.chit_value = random.choice(chit_values)
174:
175:     def _default_assign_tile_chit_values(self, start=2, end=12,
176:                                         exclude=Calamity.DEFAULT_ROLL_VALUES):
177:         """Assign chit values in a manner similar to that of the original game.
178:
179:         Specifically, find out how many times each value would occur if we
180:         were to distribute them over the board's non-fallow tiles evenly,
181:         except for the highest and lowest values (presumably the least likely
182:         to occur), which should only appear on the board half as often.
183:
184:         Args:
185:             start (int): Together with end, defines the range of possible
186:             chit values.
187:
188:             end (int): See above.
189:
190:             exclude (list): A list of values that lie in the range defined by
191:             start and end that should not be included in the set of possible
192:             chit values.
193:
194:         Returns:
195:             None.
196:
197:         TODO: Consider storing self.tile_count instead of using the length
198:         of the iterator. For now, however, performance not an issue.

```

```

199:
200:     """
201:     chit_values = filter(
202:         lambda value: value not in exclude, range(start, end + 1)
203:     )
204:
205:     min_chit_value = chit_values[0]
206:     max_chit_value = chit_values[-1]
207:
208:     # We only want to consider arable tiles.
209:     arable_tiles = list(self.iter_arable_tiles())
210:     tile_count = len(arable_tiles)
211:
212:     # Since the lowest and highest chit values will occur half as
213:     # frequently, we act as if we were only had len(chit_values) - 1 values.
214:     per_value_count = tile_count / (len(chit_values) - 1)
215:
216:     # We want the highest and lowest value chits to appear half as often.
217:     def get_value_occurrence_count(value):
218:         if value == min_chit_value or value == max_chit_value:
219:             return per_value_count / 2
220:         else:
221:             return per_value_count
222:
223:     # Get a list of all the chit values we will place e.g. if we expect
224:     # to place 5 chits of value 3, then 3 should occur 5 times in the list.
225:     chit_values_to_assign = Utils.flatten(map(
226:         lambda value: [value] * get_value_occurrence_count(value),
227:         chit_values
228:     ))
229:
230:     # Assign chit values to arable tiles only.
231:     for tile, chit_value_to_assign in zip(arable_tiles,
232:                                         chit_values_to_assign):
233:         tile.chit_value = chit_value_to_assign
234:
235:     def assign_tile_harbors(self):
236:         """Assign harbors to this board.
237:
238:         TODO: Officially, harbors seem to be placed after every
239:             3rd then 3rd then 4th edge. This is a pain to program given that
240:             it only _seems_ that way.
241:
242:         """
243:
244:         # TODO
245:         pass
246:
247:     def iter_arable_tiles(self):
248:         """Iterate over this board's non-fallow i.e. arable tiles."""
249:
250:         for tile in self.iter_tiles():
251:             if tile.resource_type != ResourceType.FALLOW:
252:                 yield tile
253:
254:     def place_vertex_structure(self, x, y, vertex_dir, structure,
255:                               must_border_claimed_edge=True, struct_x=None,
256:                               struct_y=None, struct_vertex_dir=None):
257:         """Place a structure of the given type on the specified vertex.
258:
259:         Args:
260:             See self.update_vertex().
261:
262:             structure (Structure): Structure to replace the specified vertex
263:             with.
264:
265:         Returns:

```

```

265:         None.
266:
267:     Raises:
268:         InvalidBaseStructureException. If structure to be placed is an
269:         upgrade or extension of a structure class that hasn't been
270:         placed at the defined vertex.
271:     """
272:
273:     tile = self.tiles[x][y]
274:     old_vertex_val = tile.vertices[vertex_dir]
275:
276:     self.validate_structure_placement(x, y, old_vertex_val, structure,
277:                                     vertex_dir, must_border_claimed_edge,
278:                                     struct_x, struct_y, struct_vertex_dir)
279:
280:     self.update_vertex(x, y, vertex_dir, structure)
281:
282:     def place_edge_structure(self, x, y, edge_dir, structure,
283:                             must_border_claimed_edge=True, struct_x=None,
284:                             struct_y=None, struct_vertex_dir=None):
285:         tile = self.tiles[x][y]
286:         vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)
287:         old_edge_val = tile.edges[vertex_dirs[0]][vertex_dirs[1]]
288:
289:         self.validate_structure_placement(x, y, old_edge_val, structure,
290:                                         edge_dir, must_border_claimed_edge,
291:                                         struct_x, struct_y, struct_vertex_dir)
292:
293:         self.update_edge(x, y, edge_dir, structure)
294:
295:     def validate_structure_placement(self, x, y, old_value, new_value,
296:                                     placement_dir, must_border_claimed_edge,
297:                                     struct_x, struct_y, struct_vertex_dir):
298:
299:         # A structure can only be placed on a vertex if none of the three
300:         # adjacent vertices are occupied aka the Distance Rule.
301:         if new_value.position_type == PositionType.VERTEX:
302:
303:             adjacent_vertex_vals = \
304:                 self.get_adjacent_vertices_for_vertex(x, y, placement_dir)
305:
306:             adjacent_structures = filter(
307:                 lambda vertex_val: isinstance(vertex_val, Structure),
308:                 adjacent_vertex_vals
309:             )
310:
311:             if len(adjacent_structures):
312:                 raise InvalidStructurePlacementException()
313:
314:             # If the struct_x etc. are provided, they specify a vertex the new
315:             # edge to place must border e.g. as in initial placement stage.
316:             if new_value.position_type == PositionType.EDGE and \
317:                 struct_x is not None:
318:                 allowable_edges = self.get_adjacent_edges(struct_x, struct_y, struct_ver
319: tex_dir)
320:                 target_edge = self.get_tile_with_coords(x, y).get_edge(placement_dir)
321:
322:                 if target_edge not in allowable_edges:
323:                     raise InvalidStructurePlacementException()
324:
325:             # If the player is replacing an existing structure...
326:             if isinstance(old_value, Structure):
327:
328:                 # The old structure must be owned by the same player.
329:                 if old_value.owning_player != new_value.owning_player:
330:                     raise BoardPositionOccupiedException((x, y), old_value,

```

```

330:                                     old_value.owning_player)
331:
332:         # The new value must be an augmenting structure whose base structure
333:         # matches the existing structure.
334:         if (not new_value.is_augmenting_structure()) or \
335:             (new_value.is_augmenting_structure() and \
336:              old_value.name != new_value.augments):
337:             raise InvalidBaseStructureException(old_value, new_value)
338:
339:         # If the player is not replacing an existing structure, make sure it's
340:         # neighboring a road, unless overridden e.g. as during initial
341:         # structure placement.
342:         elif must_border_claimed_edge:
343:             if placement_dir in EdgeDirection:
344:                 edge_vals = self.get_adjacent_edges_for_edge(x, y, placement_dir)
345:             elif placement_dir in VertexDirection:
346:                 edge_vals = self.get_adjacent_edges_to_vertex(x, y, placement_dir)
347:
348:             claimed_edge_structs = filter(
349:                 lambda edge_val: isinstance(edge_val, Structure) and
350:                                     edge_val.owning_player == new_value.owning_player,
351:                 edge_vals
352:             )
353:
354:             if not len(claimed_edge_structs):
355:                 raise InvalidStructurePlacementException()
356:
357:     def distribute_resources_for_roll(self, roll_value):
358:         """Distribute resources to the players based on the given roll value.
359:
360:         Resources are distributed as follows: Whenever a value is rolled that
361:         matches the chit value of a tile, for all structures on that tile,
362:         distribute the number of resources dictated by the yield of that
363:         structure of the type of that tile.
364:
365:         Args:
366:             roll_value (int): Dice roll value used to determine which tiles
367:                             should yield resources this turn.
368:
369:         Returns:
370:             dict. Primary keys are players and secondary keys are resource
371:             types. Stored values are the number of a given resource that was
372:             distributed to the player.
373:         """
374:
375:         # Find those tiles whose chit value matches the roll value,
376:         # and whose yield isn't blocked by a calamity.
377:         resource_tiles = filter(
378:             lambda tile:
379:                 tile.chit_value == roll_value and
380:                 (CalamityTilePlacementEffect.BLOCK_YIELD not in
381:                  tile.get_calamity_tile_placement_effects()),
382:             list(self.iter_tiles())
383:         )
384:
385:         distributions = Utils.nested_dict()
386:
387:         # Create a dictionary that stores per-player resource distributions.
388:         # i.e. distributions => player => resource_type => (int)
389:         for resource_tile in resource_tiles:
390:
391:             # Find any structures built on the vertices of the found tiles.
392:             adjacent_structures = resource_tile.get_adjacent_vertex_structures()
393:
394:             for structure in adjacent_structures:
395:                 player = structure.owning_player

```

```
396:         resource_type = resource_tile.resource_type
397:         resource_yield = structure.base_yield
398:
399:         if not distributions[player][resource_type]:
400:             distributions[player][resource_type] = 0
401:
402:         distributions[player][resource_type] += resource_yield
403:
404:     self.distribute_resources(distributions)
405:
406:     return distributions
407:
408: def distribute_resources(self, distributions):
409:
410:     # Now distribute resources to players, if the bank has enough.
411:     for resource_type in ResourceType.get_arable_types():
412:
413:         def get_per_player_production(player):
414:             resource_count = distributions[player][resource_type]
415:             return resource_count if resource_count else 0
416:
417:         total_count = sum(map(get_per_player_production, distributions))
418:
419:         try:
420:             self.bank.withdraw_resources(resource_type, total_count)
421:
422:             for player in distributions:
423:
424:                 count = distributions[player][resource_type]
425:
426:                 if count:
427:                     player.deposit_resources(resource_type, count)
428:
429:             except NotEnoughResourcesException:
430:                 # Bank didn't have enough of the current resource to distribute
431:                 # to all players, so distribute none of this resource.
432:                 pass
433:
434:         return distributions
435:
436: def find_robber(self):
437:     """Return the robber we can find."""
438:
439:     for tile in self.iter_tiles():
440:         for calamity in tile.calamities:
441:             if isinstance(calamity, Robber):
442:                 return calamity
443:
444:     return None
445:
446: def get_tile_of_resource_type(self, resource_type):
447:     """Returns first found file of specified resource type."""
448:
449:     for tile in self.iter_tiles():
450:         if tile.resource_type == resource_type:
451:             return tile
452:
453:     return None
454:
455: def find_tile_with_calamity(self, calamity):
456:
457:     for tile in self.iter_tiles():
458:         if calamity in tile.calamities:
459:             return tile
460:
461:     return None
```

```
462:
463:     def place_calamity(self, x, y, calamity):
464:
465:         tile = self.get_tile_with_coords(x, y)
466:         tile.add_calamity(calamity)
```



```

1: # -*- coding: utf-8 -*-
2: import pdb
3:
4: from engine.src.lib.utils import Utils
5: from engine.src.board.board import Board
6: from engine.src.tile.hex_tile import HexTile
7: from engine.src.vertex import Vertex
8: from engine.src.edge import Edge
9: from engine.src.direction.edge_direction import EdgeDirection
10: from engine.src.direction.vertex_direction import VertexDirection
11: from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping
12:
13:
14: class HexBoard(Board):
15:     """A horizontal hextile board, such as that used in Settlers of Catan.
16:
17:     Hextiles are referred to using axial coordinates.
18:     See below for more on axial hex coordinates.
19:     http://devmag.org.za/2013/08/31/geometry-with-hex-coordinates/
20:     www.redblobgames.com/grids/hexagons
21:
22:     Attributes:
23:         radius (int): The number of tiles between the center tile and the edge
24:             of the board, including the center tile itself. Should be >= 1.
25:
26:         tiles (dict): A dictionary of tiles, indexed using axial coordinates
27:
28:         tile_cls (class): Class of the tiles to be generated during board
29:             initialization.
30:
31:     Args:
32:         radius (int): The number of tiles between the center tile and the edge
33:             of the board, including the center tile itself. Should be >= 1.
34:
35:     """
36:     MIN_BOARD_RADIUS = 1
37:
38:     def __init__(self, radius, tile_cls=HexTile):
39:
40:         if radius < HexBoard.MIN_BOARD_RADIUS:
41:             message = ("Specified radius does not meet the minimum board "
42:                        "tile radius {0}").format(HexBoard.MIN_BOARD_RADIUS)
43:             raise ValueError(message)
44:
45:         self.radius = radius
46:
47:         self.tile_cls = tile_cls
48:
49:         self.tiles = {}
50:         self._create_tiles()
51:
52:     def _create_tiles(self):
53:         """Generates a dictionary of tiles, indexed by axial coordinates.
54:
55:         See how coordinates are generated in _add_new_tile_with_coords()
56:
57:         Returns:
58:             None.
59:
60:         """
61:         for x, y in self.iter_tile_coords():
62:             self._add_new_tile_with_coords(x, y)
63:
64:         self._sync_tile_vertices_and_edges()
65:
66:

```

```

67:
68:     def _add_new_tile_with_coords(self, x, y):
69:         """Add a brand new tile to the board at the given axial coordinates."""
70:
71:         if x not in self.tiles:
72:             self.tiles[x] = {}
73:
74:         tile = self.tile_cls(x, y)
75:         self.tiles[x][y] = tile
76:
77:     def _sync_tile_vertices_and_edges(self):
78:         """Synchronize shared vertices and edges across tiles.
79:
80:         New tile objects will create their own vertices and edges. When tiles
81:         share edges and vertices with existing tiles on the board, however,
82:         we want them to point to the same shared vertex or edge objects,
83:         instead of each having their own. This method enforces this for the
84:         given tile.
85:
86:         """
87:         for x, y in self.iter_tile_coords():
88:             tile = self.get_tile_with_coords(x, y)
89:
90:             for vertex_dir in VertexDirection:
91:                 new_vertex = Vertex()
92:                 self.update_vertex(x, y, vertex_dir, new_vertex)
93:
94:             for edge_dir in EdgeDirection:
95:                 new_edge = Edge()
96:                 self.update_edge(x, y, edge_dir, new_edge)
97:
98:     def get_tile_with_coords(self, x, y):
99:         """Get the tile at the given coordinates, or None if no tile exists."""
100:
101:         if x in self.tiles and y in self.tiles[x]:
102:             return self.tiles[x][y]
103:
104:         return None
105:
106:     def get_vertex(self, x, y, vertex_dir):
107:         """Get the vertex defined by the given params."""
108:         tile = self.get_tile_with_coords(x, y)
109:
110:         if tile:
111:             return tile.vertices[vertex_dir]
112:         else:
113:             return None
114:
115:     def valid_tile_coords(self, x, y):
116:         """Return whether or not these params specify a valid tile."""
117:
118:         return bool(self.get_tile_with_coords(x, y))
119:
120:     def valid_vertex(self, x, y, vertex_dir):
121:         """Return whether or not these params specify a valid vertex."""
122:
123:         return bool(self.get_vertex(x, y, vertex_dir))
124:
125:     def get_neighboring_tile(self, tile, edge_direction):
126:         """Get the tile neighboring the given tile in the given direction.
127:
128:         Args:
129:             tile (Tile): The tile for which we'd like to find the neighbor.
130:
131:             edge_direction (EdgeDirection): Hextiles have 6 edges and thus
132:                 neighbors in 6 different directions. Should be relative to the
133:                 given tile.

```

```

133:
134:     Returns:
135:         Tile. None if the tile has no valid neighbor in that direction.
136:
137:     TODO: enforce that direction is actually in EdgeDirection
138:     """
139:
140:     x = tile.x + edge_direction[0]
141:     y = tile.y + edge_direction[1]
142:
143:     return self.get_tile_with_coords(x, y)
144:
145: def get_neighboring_tiles(self, tile):
146:     """Get all six neighboring tiles for the given hextile.
147:
148:     Args:
149:         tile (Tile): The tile whose neighbors we want to return.
150:
151:     Returns:
152:         dict. Keys are directions and values are tiles that neighbor the
153:         given tile in that direction.
154:     """
155:
156:     neighboring_tiles = {}
157:
158:     for direction in EdgeDirection:
159:         neighbor_tile = self.get_neighboring_tile(tile, direction)
160:
161:         if neighbor_tile:
162:             neighboring_tiles[direction] = neighbor_tile
163:
164:     return neighboring_tiles
165:
166: def iter_tiles(self):
167:     """Iterate over the tiles in this board.
168:
169:     The order is that described in iter_tile_coords.
170:
171:     Yields:
172:         Tile. Each tile of the board.
173:     """
174:
175:     for x, y in self.iter_tile_coords():
176:         yield self.get_tile_with_coords(x, y)
177:
178: def iter_perimeter_tiles(self):
179:     """Iterate over the tiles along the outermost edge of the board."""
180:
181:     for x, y in HexBoard.iter_tile_ring_coords(self.radius - 1):
182:         yield self.get_tile_with_coords(x, y)
183:
184: def iter_tile_coords(self):
185:     """Iterate over axial coordinates for each tile in the board.
186:
187:     This is a generator function that will yield the coordinates to the
188:     caller each time after they are computed.
189:
190:     We can consider a hextile board a series of concentric rings where the
191:     radius counts the number of concentric rings that compose the board.
192:     When generating coordinates, we traverse each such ring one at a time,
193:     using the pattern specified in iter_tile_ring_coords().
194:
195:     Yields:
196:         tuple. The axial (x, y) coordinates of each tile on the board.
197:     """
198:
199:     for ring_index in range(self.radius):

```

```

199:         for x, y in HexBoard.iter_tile_ring_coords(ring_index):
200:             yield x, y
201:
202: @staticmethod
203: def iter_tile_ring_coords(ring_index):
204:     """Iterate clockwise over coordinates of the board's perimeter tiles.
205:
206:     We can consider a hextile board a series of concentric rings where the
207:     radius counts the number of concentric rings that compose the board.
208:     Thus, ring_index 0 corresponds to the center tile and ring_index =
209:     self.radius - 1 corresponds to perimeter tiles.
210:
211:     Here we generate the coordinates for all tiles of a single ring,
212:     designated by ring_index, traversing the ring one tile at a time,
213:     starting from the westernmost tile and continuing around the ring in a
214:     clockwise fashion.
215:
216:     Args:
217:         ring_index (int): Defines which tile ring to iterate over.
218:         Should be a value between 0 and self.radius - 1.
219:
220:     Yields:
221:         tuple. The axial (x, y) coordinates of each tile in the given ring.
222:     """
223:
224:     # We start yielding coordinates from the westernmost tile.
225:     x = -1 * ring_index
226:     y = 0
227:
228:     if x == 0 and y == 0:
229:         yield x, y
230:
231:     # First we scale the northwest side of the ring.
232:     # This is equivalent to moving along the y-axis of the board.
233:     while y != ring_index:
234:         yield x, y
235:         y += 1
236:
237:     # Then we scale the northern side of the ring.
238:     # This is equivalent to moving along the x-axis of the board.
239:     while x != 0:
240:         yield x, y
241:         x += 1
242:
243:     # Then we scale the northeast side of the ring.
244:     # This is equivalent to moving along the z-axis of the board.
245:     while x != ring_index or y != 0:
246:         yield x, y
247:         x += 1
248:         y -= 1
249:
250:     # Then we scale the southeast side of the ring.
251:     while y != -ring_index:
252:         yield x, y
253:         y -= 1
254:
255:     # Then the south side of the ring.
256:     while x != 0:
257:         yield x, y
258:         x -= 1
259:
260:     # And finally the south west side of the ring.
261:     while x != -ring_index:
262:         yield x, y
263:         x -= 1
264:         y += 1

```

```

265:
266: def update_edge(self, x, y, edge_dir, edge_val):
267:     """Update the specified edge.
268:
269:     Also updates equivalent edge for neighboring tile.
270:
271:     Args:
272:         x (int): Axial x-coordinate of the tile, one of whose vertices
273:             we will update.
274:
275:         y (int): Axial y-coordinate of the tile, one of whose vertices
276:             we will update.
277:
278:         edge_dir (EdgeDirection): Direction of edge to update relevant to
279:             tile given by x, y coordinates.
280:
281:         edge_val (Structure): Value to replace old edge values.
282:
283:     Returns:
284:         None
285:     """
286:     tile = self.get_tile_with_coords(x, y)
287:     vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)
288:
289:     neighbor_tile = self.get_neighboring_tile(tile, edge_dir)
290:
291:     tile.add_edge(vertex_dirs[0], vertex_dirs[1], edge_val)
292:
293:     # Perimeter tiles will not have neighbors along certain edges.
294:     if neighbor_tile:
295:         nv_dir_1 = HexTile.get_equivalent_vertex_dir(vertex_dirs[0], edge_dir)
296:         nv_dir_2 = HexTile.get_equivalent_vertex_dir(vertex_dirs[1], edge_dir)
297:         neighbor_tile.add_edge(nv_dir_1, nv_dir_2, edge_val)
298:
299: def update_vertex(self, x, y, vertex_dir, vertex_val):
300:     """Update the value at the specified vertex location.
301:
302:     Also updates vertex for neighboring tiles.
303:
304:     Args:
305:         x (int): Axial x-coordinate of the tile, one of whose vertices
306:             we will update.
307:
308:         y (int): Axial y-coordinate of the tile, one of whose vertices
309:             we will update.
310:
311:         vertex_dir (VertexDirection): Vertex direction, relative to the
312:             tile specified by the x and y coordinates, of the vertex to
313:             update.
314:
315:         vertex_val (Structure): Value to replace old vertex values.
316:
317:     Returns:
318:         None.
319:     """
320:
321:     tile = self.get_tile_with_coords(x, y)
322:     old_vertex_val = self.get_vertex(x, y, vertex_dir)
323:
324:     tile.vertices[vertex_dir] = vertex_val
325:
326:     # Get the two edges of the found tile that have as an endpoint
327:     # a vertex of the given vertex direction.
328:     vertex_adj_edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(
329:         vertex_dir)
330:

```

```

331:     for vertex_adj_edge_dir in vertex_adj_edge_dirs:
332:         neighbor_tile = self.get_neighboring_tile(tile, vertex_adj_edge_dir)
333:
334:         # Edge tiles may not have neighboring tiles in the given direction.
335:         if neighbor_tile:
336:             neighbor_vertex_dir = HexTile.get_equivalent_vertex_dir(
337:                 vertex_dir, vertex_adj_edge_dir)
338:
339:             neighbor_tile.update_vertex(neighbor_vertex_dir, vertex_val)
340:
341: def get_adjacent_tiles_to_vertex(self, x, y, vertex_dir):
342:     """Get the three tiles that converge at the specified vertex.
343:
344:     Args:
345:         x (int): Axial x-coordinate of the tile, one of whose vertices
346:             we will update.
347:
348:         y (int): Axial y-coordinate of the tile, one of whose vertices
349:             we will update.
350:
351:         vertex_dir (VertexDirection): Vertex direction, relative to the
352:             tile specified by the x and y coordinates, of the vertex to
353:             find the adjacent tiles of.
354:
355:     Returns:
356:         list of Tiles. The tiles that converge at the specified vertex.
357:     """
358:
359:     tile = self.get_tile_with_coords(x, y)
360:
361:     adjacent_tiles = map(
362:         lambda edge_dir: self.get_neighboring_tile(tile, edge_dir),
363:         EdgeVertexMapping.get_edge_dirs_for_vertex_dir(vertex_dir)
364:     )
365:
366:     adjacent_tiles.append(tile)
367:
368:     return adjacent_tiles
369:
370: def get_adjacent_edges(self, x, y, vert_or_edge_dir, return_values=True):
371:     if vert_or_edge_dir in EdgeDirection:
372:         if return_values:
373:             return self.get_adjacent_edges_for_edge(x, y, vert_or_edge_dir)
374:         else:
375:             return self._get_adjacent_edges_for_edge(x, y, vert_or_edge_dir)
376:
377:     elif vert_or_edge_dir in VertexDirection:
378:         if return_values:
379:             return self.get_adjacent_edges_to_vertex(x, y, vert_or_edge_dir)
380:         else:
381:             return self._get_adjacent_edges_to_vertex(x, y, vert_or_edge_dir)
382:
383: def _get_adjacent_edges_to_vertex(self, x, y, vertex_dir):
384:
385:     tile = self.get_tile_with_coords(x, y)
386:
387:     edge_vals = []
388:
389:     # Get the directions of edges that both have vertex_dir as an endpoint.
390:     edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(vertex_dir)
391:
392:     edge_vals.append( (x, y, edge_dirs[0]) )
393:     edge_vals.append( (x, y, edge_dirs[1]) )
394:
395:     # The last edge value won't be available via the current tile's edges,
396:     # but must be found on its neighbor.

```

```

397:         neighbor_x = tile.x + edge_dirs[0][0]
398:         neighbor_y = tile.y + edge_dirs[0][1]
399:         neighboring_tile = self.get_neighboring_tile(tile, edge_dirs[0])
400:         opp_vert_dir = HexTile.get_equivalent_vertex_dir(vertex_dir, edge_dirs[0])
401:
402:         neighbor_edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(opp_vert
_dir)
403:         neighbor_edge_dir = next(d for d in neighbor_edge_dirs if d not in \
404:             map(lambda edge_val: edge_val[2].get_opposite_direction(), edge_vals))
405:
406:         edge_vals.append( (neighbor_x, neighbor_y, neighbor_edge_dir) )
407:
408:         return edge_vals
409:
410:     def get_adjacent_edges_to_vertex(self, x, y, vertex_dir):
411:
412:         edge_tuples = self._get_adjacent_edges_to_vertex(x, y, vertex_dir)
413:         edge_vals = []
414:
415:         msg = "Edges adjacent to ({}, {}) {}: \n".format(x, y, vertex_dir)
416:
417:         for x, y, edge_dir in edge_tuples:
418:             tile = self.get_tile_with_coords(x, y)
419:             edge_val = tile.get_edge(edge_dir)
420:
421:             edge_vals.append(edge_val)
422:             msg += '\t\t({}, {}) {} \n'.format(x, y, edge_dir)
423:
424:         return edge_vals
425:
426:     def _get_adjacent_edges_for_edge(self, x, y, edge_dir):
427:
428:         vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)
429:
430:         edge_tuples = []
431:         edge_tuples.extend(self._get_adjacent_edges_to_vertex(x, y, vertex_dirs[0])
+ \
432:             self._get_adjacent_edges_to_vertex(x, y, vertex_dirs[1]))
433:
434:         edge_tuples = filter(
435:             lambda edge_tuple: edge_tuple[2] != edge_dir,
436:             edge_tuples
437:         )
438:
439:         return edge_tuples
440:
441:     def get_adjacent_edges_for_edge(self, x, y, edge_dir):
442:
443:         edge_tuples = self._get_adjacent_edges_for_edge(x, y, edge_dir)
444:         edge_vals = []
445:
446:         for ex, ey, e_dir in edge_tuples:
447:             tile = self.get_tile_with_coords(ex, ey)
448:
449:             if tile:
450:                 edge_vals.append(tile.get_edge(e_dir))
451:
452:         return edge_vals
453:
454:     def _get_adjacent_vertices_for_vertex(self, x, y, vertex_dir):
455:
456:         vertex_tuples = []
457:
458:         tile = self.get_tile_with_coords(x, y)
459:
460:         vertex_dirs = VertexDirection.get_neighboring_vertex_dirs(vertex_dir)
461:
462:         # Two of the closest vertices will lie on this tile
463:         for adjacent_vertex_dir in vertex_dirs:
464:             vertex_tuple = (x, y, adjacent_vertex_dir)
465:             vertex_tuples.append(vertex_tuple)
466:
467:         # The last vertex value won't be available via the current tile's
468:         # vertices, but must be found on its neighbor.
469:
470:         edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(vertex_dir)
471:
472:         # Pick one edge, arbitrarily, to find the neighbor tile relative to that edg
e.
473:         neighbor_edge_dir = edge_dirs[0]
474:         neighboring_tile = self.get_neighboring_tile(tile, neighbor_edge_dir)
475:         neighbor_x = tile.x + neighbor_edge_dir[0]
476:         neighbor_y = tile.y + neighbor_edge_dir[1]
477:
478:         # Find the neighbor equivalent of vertex_dir
479:         opp_vert_dir = HexTile.get_equivalent_vertex_dir(vertex_dir, neighbor_edge_d
ir)
480:
481:         # Vertex and edge direction should be relative to same tile
482:         def vertex_already_found(v_dir, neighbor_edge):
483:             neighbor_equivalent_v_dir = \
484:                 HexTile.get_equivalent_vertex_dir(v_dir, neighbor_edge_dir.get_oppo
site_direction()),
485:                 return neighbor_equivalent_v_dir not in map(lambda v_tup: v_tup[2], vert
ex_tuples)
486:
487:         # Find the vertices adjacent to neighbors equivalent of vertex_dir.
488:         # One will duplicate a vertex we already have, one will be new.
489:         # Filter out the duplicate.
490:         last_vertex_dir = filter(
491:             lambda v_dir: not vertex_already_found(v_dir, neighbor_edge_dir.get_oppo
site_direction()),
492:             VertexDirection.get_neighboring_vertex_dirs(opp_vert_dir)
493:         )
494:
495:         if len(last_vertex_dir):
496:             last_vertex_dir = last_vertex_dir[0]
497:             vertex_tuples.append( (neighbor_x, neighbor_y, last_vertex_dir) )
498:
499:         return vertex_tuples
500:
501:     def get_adjacent_vertices_for_vertex(self, x, y, vertex_dir):
502:
503:         vertex_tuples = self._get_adjacent_vertices_for_vertex(x, y, vertex_dir)
504:         vertex_vals = []
505:
506:         for vx, vy, v_dir in vertex_tuples:
507:             tile = self.get_tile_with_coords(vx, vy)
508:
509:             if tile:
510:                 vertex_vals.append(tile.get_vertex(v_dir))
511:
512:         return vertex_vals

```

```
1: # -*- coding: utf-8 -*-
2: from abc import ABCMeta, abstractmethod, abstractproperty
3: from enum import Enum
4:
5:
6: class Calamity(object):
7:     """
8:     TODO: Consider breaking Calamity subclasses based on their latent effect,
9:     i.e. when not rolled, but on the board. So robbers block tile yield.
10:    Other calamities might block structure construction.
11:    """
12:    __metaclass__ = ABCMeta
13:
14:    DEFAULT_ROLL_VALUES = [7]
15:
16:    @abstractproperty
17:    def roll_value(self):
18:        """The dice roll value that should trigger this calamity's effect."""
19:        pass
20:
21:    @abstractmethod
22:    def trigger_effect(self, game, player):
23:        """Activates this calamity's effect.
24:
25:        Args:
26:            game (Game): The game this calamity will affect.
27:
28:            player (Player): Player who rolled the triggering roll.
29:        """
30:        pass
31:
32:
33: class CalamityTilePlacementEffect(Enum):
34:     BLOCK_YIELD = 1
```



```

1: # -*- coding: utf-8 -*-
2: from engine.src.calamity.calamity import Calamity
3: from engine.src.calamity.calamity import CalamityTilePlacementEffect
4:
5:
6: class Robber(Calamity):
7:
8:     MIN_ROBBER_ACTIVATING_RESOURCE_COUNT_THRESHOLD = 8
9:
10:    def __init__(self):
11:        # TODO: Not sure if this is the best way to represent these effects.
12:        self.tile_placement_effect = CalamityTilePlacementEffect.BLOCK_YIELD
13:
14:    def roll_value(self):
15:        # TODO: Move to config?
16:        return 7
17:
18:    def trigger_effect(self, game, player):
19:        """Halve players resources, move the robber, draw a resource card.
20:
21:        Triggering the robber effect elicits the following behavior:
22:        (1) All players who have more than some threshold of resource cards
23:            must discard half of their resource hand, floored.
24:        (2) See self.outside_trigger_effect().
25:
26:        Args:
27:            See Calamity.
28:        """
29:
30:        threshold = Robber.MIN_ROBBER_ACTIVATING_RESOURCE_COUNT_THRESHOLD
31:
32:        # Have players discard half their hand if they have too many cards.
33:        for game_player in game.players:
34:
35:            resource_count = game_player.count_resources()
36:
37:            if resource_count > threshold:
38:                cards_to_discard = int(resource_count / 2)
39:                resources = game_player.get_resource_list()
40:
41:                resource_indices = game.input_manager.prompt_discard_resources(
42:                    game, player, resources, cards_to_discard)
43:
44:                for index in resource_indices:
45:                    game_player.withdraw_resources(resources[index], 1)
46:
47:            self.outside_trigger_effect(game, player)
48:
49:    def outside_trigger_effect(self, game, player):
50:        """When the robber is activated not by a dice roll, call this method.
51:
52:        Execute the following behavior:
53:        (1) The robber should be moved to a different tile.
54:        (2) A resource card must be drawn from one of the players with
55:            structures built adjacent to the tile.
56:        """
57:
58:        robber_successfully_moved = False
59:        previous_tile = game.board.find_tile_with_calamity(self)
60:        previous_tile.remove_calamity(self)
61:
62:        tile = None
63:
64:        prompt = 'Select a tile to move the robber to. Current location: {0}'\
65:            .format(previous_tile)
66:

```

```

67:        game.input_manager.input_default(prompt, None, False)
68:
69:        while not robber_successfully_moved:
70:            x, y = game.input_manager.prompt_tile_coordinates(game)
71:
72:            # Move robber to new tile.
73:            tile = game.board.get_tile_with_coords(x, y)
74:
75:            if tile != previous_tile:
76:                tile.add_calamity(self)
77:                robber_successfully_moved = True
78:
79:        # Draw card from player that has a structure built adjacent to the tile.
80:        # The player can not draw from herself or from a player with no cards.
81:        eligible_players = filter(
82:            lambda owning_player:
83:                owning_player != player and
84:                owning_player.count_resources() != 0,
85:            map(lambda structure: structure.owning_player,
86:                tile.get_adjacent_vertex_structures()))
87:
88:
89:        if eligible_players:
90:
91:            # Chose a player to randomly select a resource from.
92:            chosen_player = game.input_manager.prompt_select_player(
93:                game, eligible_players)
94:
95:            resource_type = chosen_player.withdraw_random_resource()
96:            player.deposit_resources(resource_type, 1)
97:
98:            # Announce received resource.
99:            msg = 'You received 1 {0} from {1}.'.format(
100:                resource_type, chosen_player.name)
101:            game.input_manager.input_default(msg, None, False)
102:
103:        else:
104:            # Announce no eligible players to draw from.
105:            msg = 'No qualifying players to draw from.'
106:            game.input_manager.input_default(msg, None, False)

```



```
1: # -*- coding: utf-8 -*-
2: from engine.src.config.config import Config
3: from engine.src.lib.utils import Utils
4:
5:
6: class DevelopmentCard(object):
7:     """
8:     Attributes:
9:         From Config:
10:             count (int)
11:             name (str)
12:             description (str)
13:             draw_card (func)
14:             play_card (func)
15:             cost (int)
16:
17:             played (bool)
18:             is_playable (bool)
19:     """
20:
21:     def __init__(self, **kwargs):
22:
23:         # Initialize default values.
24:         Config.init_from_config(self, 'game.card.development.default')
25:
26:         # Overwrite default values with custom values.
27:         Utils.init_from_dict(self, kwargs)
28:
29:         self.played = False
30:         self.is_playable = True
31:
32:     def __str__(self):
33:         return self.name
34:
35:     def draw_card(self, game, player):
36:         """Draw this card and activate any effect incurred by holding it.
37:
38:         This method should be called only once when purchased by a player.
39:
40:         Args:
41:             game (Game): The game this card may possibly affect.
42:
43:             player (Player): The player that bought this development card.
44:
45:         Returns:
46:             None. Should call functions on game and player.
47:         """
48:         pass
49:
50:     def play_card(self, game, player):
51:         """Draw this card and activate any relevant effect.
52:
53:         This method should be called only once when played by a player.
54:
55:         Args:
56:             game (Game): The game this card may possibly affect.
57:
58:             player (Player): The player that played this development card.
59:
60:         Returns:
61:             None. Should call functions on game and player.
62:         """
63:
64:         self.played = True
```



```
1: def draw_card(self, game, player):
2:     pass
3:
4:
5: def play_card(self, game, player):
6:     """Move the robber and draw a card from another adjacent player."""
7:
8:     game.input_manager.announce_development_card_played(player, self)
9:
10:    robber = game.board.find_robber()
11:
12:    robber.outside_trigger_effect(game, player)
13:
14:    player.knights += 1
15:
16:    self.played = True
```



```
1: def draw_card(self, game, player):
2:     pass
3:
4:
5: def play_card(self, game, player):
6:     """Allow player to take all carried cards of selected resource type."""
7:
8:     game.input_manager.announce_development_card_played(player, self)
9:     resource_type = game.input_manager.prompt_select_resource_type()
10:
11:     for game_player in game.players:
12:         if player != game_player:
13:             count = player.resources[resource_type]
14:
15:             game_player.transfer_resources(player, resource_type, count)
16:
17:             msg = '{0} received {1} {2} from {3}'.format(
18:                 player.name, count, resource_type, game_player.name)
19:
20:             game.input_manager.input_default(msg, None, False)
21:
22:     # Announce finished collecting resources.
23:     msg = 'Done monopolizing resources.'
24:     game.input_manager.input_default(msg, None, False)
25:
26:     self.played = True
```



```
1: def draw_card(self, game, player):
2:     pass
3:
4:
5: def play_card(self, game, player):
6:     """Allow player to take all carried cards of selected resource type."""
7:
8:     game.input_manager.announce_development_card_played(player, self)
9:
10:    for _ in range(2):
11:        x, y, edge_dir = game.input_manager.prompt_edge_placement(game)
12:        game.board.place_edge_structure(x, y, edge_dir,
13:                                       player.get_structure('road'))
14:
15:    self.played = True
```



```
1: def draw_card(self, game, player):
2:     player.hidden_points += 1
3:
4:
5: def play_card(self, game, player):
6:     # We could convert the player's hidden points to public points,
7:     # but keeping the points hidden makes it easier to recompute
8:     # a player's overall point total from scratch.
9:     pass
```



```
1: def draw_card(self, game, player):
2:     pass
3:
4:
5: def play_card(self, game, player):
6:     """Allow player to take 2 cards of their chosen resource type."""
7:
8:     game.input_manager.announce_development_card_played(player, self)
9:     resource_type = game.input_manager.prompt_select_resource_type()
10:
11:     game.board.bank.transfer_resources(player, resource_type, 2)
12:
13:     self.played = True
```



```

1: from types import *
2: from engine.src.lib.utils import Utils
3: from engine.src.config.game_config import game_config
4: from engine.src.config.type_config import type_config
5: from engine.src.config.type_mapping import type_mapping
6: from engine.src.exceptions import *
7: import pdb
8:
9:
10: class Config(object):
11:
12:     is_coerced = False
13:
14:     @classmethod
15:     def init_from_config(cls, obj, config_path):
16:         property_dict = Config.get(config_path)
17:         dct = { Utils.convert_format(k): v for (k, v) in property_dict.iteritems() }
18:         Utils.init_from_dict(obj, dct)
19:
20:     @classmethod
21:     def pluck(cls, config_path, prop):
22:         target_dict = Config.get(config_path)
23:         return Utils.pluck(target_dict, prop, True)
24:
25:     @classmethod
26:     def set(cls, value, dot_notation_str, dct=None):
27:
28:         if dct is None:
29:             dct = Config.config
30:
31:         keys = dot_notation_str.split('.')
32:
33:         def set_recursive(dct, keys):
34:             if not keys:
35:                 return dct
36:
37:             key = keys.pop(0)
38:             val = None
39:
40:             if key in dct:
41:                 val = dct.get(key)
42:             else:
43:                 raise NoConfigValueDefinedException(dot_notation_str)
44:
45:             # If we still have keys left, the property we want to set is nested
46:             # somewhere inside the value we fetched.
47:             if keys:
48:                 if val:
49:                     return set_recursive(val, keys)
50:                 else:
51:                     raise NoConfigValueDefinedException(dot_notation_str)
52:             # If we have no keys left, we've found the target value.
53:             else:
54:                 dct[key] = value
55:
56:         set_recursive(dct, keys)
57:
58:     @classmethod
59:     def get(cls, dot_notation_str, dct=None, remove_default=True):
60:         """Get a value from the main config dict given a dot notation string.
61:
62:
63:         E.g. if caller wants config['game']['points_to_win'], they can pass in
64:         as their dot_notation_str 'game.points_to_win'.
65:
66:         See coerce() for effect of coerce_type flag.

```

```

67:
68:
69:     if not Config.is_coerced:
70:         Config.coerce_all()
71:
72:     if dct is None:
73:         dct = Config.config
74:
75:     if not dot_notation_str:
76:         return dct
77:
78:     keys = dot_notation_str.split('.')
79:
80:     def get_recursive(dct, keys):
81:         key = keys.pop(0)
82:         val = None
83:
84:         # Get the value of the key if it's in the dict.
85:         if key in dct:
86:             val = dct.get(key)
87:         elif key.replace('_', '-') in dct:
88:             val = dct.get(key.replace('_', '-'))
89:         else:
90:             # print "loc: {} \ndct: {} \nkey: {}".format(dot_notation_str, dct, ke
y)
91:             # print Config.config
92:             raise NoConfigValueDefinedException(dot_notation_str)
93:
94:         # If we still have keys left, the property we want is nested
95:         # somewhere inside the value we fetched.
96:         if keys:
97:             if val:
98:                 return get_recursive(val, keys)
99:             else:
100:                 raise NoConfigValueDefinedException(dot_notation_str)
101:         # If we have no keys left, we've found the target value.
102:         else:
103:             return val
104:
105:     value = get_recursive(dct, keys)
106:
107:     if remove_default:
108:         # Remove default value from dictionary type return value.
109:         if type(value) is dict:
110:             value = {k: value[k] for k in value.keys() if k != 'default'}
111:
112:     return value
113:
114:     @classmethod
115:     def init(cls):
116:         Config.convert_keys()
117:         Config.coerce_all()
118:
119:     @classmethod
120:     def convert_keys(cls):
121:
122:     def convert(dct):
123:         for k, v in dct.iteritems():
124:
125:             if type(k) is StringType:
126:                 dct.pop(k)
127:                 dct[Utils.convert_format(k)] = v
128:
129:             if type(v) is dict:
130:                 convert(v)
131:

```

```

132:         convert(Config.config)
133:
134:     @classmethod
135:     def coerce_all(cls):
136:         Config.is_coerced = True
137:         Config.coerce_recursive('')
138:
139:     @classmethod
140:     def coerce_recursive(cls, path_so_far):
141:         curr_value = Config.get(path_so_far, Config.config, False)
142:
143:         try:
144:             target_type = Config.get(
145:                 Config.get_default_path(path_so_far), Config.type_config, False)
146:         except NoConfigValueDefinedException:
147:             return
148:
149:         is_struct = False
150:
151:         if type(curr_value) is dict:
152:             is_struct = len(filter(
153:                 lambda key: type(key) == StringType,
154:                 target_type.keys()
155:             )) != 0
156:
157:         if is_struct:
158:             for k, v in curr_value.iteritems():
159:                 path = k if not path_so_far else '.'.join([path_so_far, k])
160:                 Config.coerce_recursive(path)
161:         else:
162:             # print "Beginning coercion, path: {}".format(path_so_far)
163:             # print "Current type: {}".format(type(curr_value))
164:             # print "Target type: {}".format(target_type)
165:             Config.set(
166:                 Config.coerce(curr_value, type(curr_value), target_type),
167:                 path_so_far
168:             )
169:
170:     @classmethod
171:     def coerce(cls, value, from_type, to_type):
172:
173:         if from_type == to_type:
174:             return value
175:
176:         if from_type is dict:
177:             result = {}
178:
179:             target_k_type = to_type.keys()[0]
180:             target_v_type = to_type.values()[0]
181:
182:             for k, v in value.iteritems():
183:                 coerced_k_value = Config.coerce(k, type(k), target_k_type)
184:                 coerced_v_value = Config.coerce(v, type(v), target_v_type)
185:
186:                 result[coerced_k_value] = coerced_v_value
187:
188:             return result
189:         else:
190:             coercion_func = type_mapping[from_type][to_type]
191:             return coercion_func(value)
192:
193:     @classmethod
194:     def get_default_path(cls, dot_notation_str):
195:         # e.g. structure.player_built.road.cost =>
196:         #       structure.player_built.default.cost
197:         """

```

```

198:         If last prop is not a dict, replace second to last with default
199:         If last prop is a dict, e.g. structure.player_built.road
200:             if dict is a struct, replace last with default
201:             if dict isn't a struct, replace second to last with default
202:         """
203:
204:         value = None
205:         path = None
206:
207:         repl_index = -1
208:
209:         while True:
210:             keys = dot_notation_str.split('.')
211:
212:             try:
213:                 keys[repl_index] = 'default'
214:                 path = '.'.join(keys)
215:                 value = Config.get(path, Config.type_config, False)
216:                 break
217:             except NoConfigValueDefinedException:
218:                 repl_index -= 1
219:             except IndexError:
220:                 # No defaults; return as is.
221:                 path = dot_notation_str
222:                 break
223:
224:         return path
225:
226:         # The dictionary accessed by Config.get()
227:         config = {}
228:
229:         # A dictionary telling us what object types we should expect
230:         # for values in config.
231:         type_config = type_config
232:
233:         type_mapping = type_mapping

```

```

63:
64:
65:
66:
-card'),
67:
-card'),
68:
69:
70:
71:
72:
73:
74:
75:
76:
d'),
77:
d'),
78:
79:
80:
81:
82:
83:
84:
85:
86:
87:
88:
rd'),
89:
90:
rd'),
91:
92:
93:
94:
95:
96:
97:
98:
99:
100:
101:
102:
103:
104:
105:
106:
107:
108:
109:
es?
110:
111:
112:
113:
114:
115:
116:
117:
118:
119:
120:
121:

```

```

        '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
    'play_card': get_import_value('card.development.monopoly', 'play

},
'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_car
    'play_card':
        get_import_value('card.development.road_building', 'play_car

},
'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_ca
    'play_card':
        get_import_value('card.development.year_of_plenty', 'play_ca

}

structures
structure': {
    layer_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 nam
            'extends': None,
            'upgrades': None,
            'position_type': 'vertex'
        },
        # Edge Structures
        'road': {
            'name': 'Road',
            'cost': {
                'lumber': 1,
                'brick': 1,
            },
            'count': 15,

```

```
122:         'point_value': 0,
123:         'base_yield': 0,
124:         'extends': None,
125:         'upgrades': None,
126:         'position_type': 'edge'
127:     },
128:     # Vertex Structures
129:     'settlement': {
130:         'name': 'Settlement',
131:         'cost': {
132:             'lumber': 1,
133:             'brick': 1,
134:             'wool': 1,
135:             'grain': 1
136:         },
137:         'count': 5,
138:         'point_value': 1,
139:         'base_yield': 1,
140:         'extends': None,
141:         'upgrades': None,
142:         'position_type': 'vertex'
143:     },
144:     'city': {
145:         'name': 'City',
146:         'cost': {
147:             'grain': 2,
148:             'ore': 3,
149:         },
150:         'count': 5,
151:         'point_value': 2,
152:         'base_yield': 2,
153:         'extends': None,
154:         'upgrades': 'Settlement',
155:         'position_type': 'vertex'
156:     },
157:     # For Demo
158:     'castle': {
159:         'name': 'Castle',
160:         'cost': {
161:             'ore': 5
162:         },
163:         'count': 2,
164:         'point_value': 3,
165:         'base_yield': 3,
166:         'extends': None,
167:         'upgrades': 'City',
168:         'position_type': 'vertex'
169:     }
170: }
171: }
172: }
173: }
```



```
1: from engine.src.resource_type import ResourceType
2: from engine.src.position_type import PositionType
3: from types import *
4:
5: type_config = {
6:     'game': {
7:         'points_to_win': IntType,
8:         'player_count': IntType,
9:
10:         'board' : {
11:             'tile_count': IntType,
12:             'radius': IntType,
13:         },
14:         'structure': {
15:             'player_built': {
16:                 'default': {
17:                     'cost': {ResourceType: IntType},
18:                     'position_type': PositionType
19:                 }
20:             }
21:         },
22:         'card': {
23:             'development': {
24:                 'default': {
25:                     'cost': {ResourceType: IntType},
26:                     'draw_card': FunctionType,
27:                     'play_card': FunctionType
28:                 }
29:             }
30:         }
31:     }
32: }
```



```
1: import engine.src.lib.utils as utils
2: from engine.src.resource_type import ResourceType
3: from engine.src.position_type import PositionType
4: from types import *
5:
6:
7: type_mapping = { # from_type => to_type => conversion function
8:     StringType: {
9:         ResourceType: lambda st: ResourceType.find_by_value(st),
10:        PositionType: lambda st: PositionType.find_by_value(st)
11:    },
12:    NoneType: {
13:        FunctionType: lambda _: utils.noop,
14:        MethodType: lambda _: utils.Utils.noop
15:    }
16: }
```



```
1: # -*- coding: utf-8 -*-
2: import random
3:
4:
5: class Dice(object):
6:     """ Represents a set of game dice.
7:
8:     Args:
9:         dice_count (int): Number of dice in the game.
10:
11:         range (list): List of possible dice values.
12:     """
13:
14:     def __init__(self, dice_count=2, values=range(1, 7)):
15:         self.dice_count = dice_count
16:         self.values = values
17:
18:     def roll(self):
19:         """ Rolls dice.
20:
21:         Returns:
22:             int. Sum of dice face values after a random throw.
23:         """
24:
25:         return sum(random.choice(self.values) for _ in range(self.dice_count))
```


`./engine/src/direction/__init__.py`

Mon Mar 30 14:08:53 2015

1

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



```
1: # -*- coding: utf-8 -*-
2: from enum import Enum
3:
4:
5: class Direction(Enum):
6:     """An abstract class that defines basic functions needed by direction enums.
7:
8:     TODO: Enforce that this class is an abstract class by having
9:           its metaclass be ABCMeta. This seems to create some issues since
10:          Enum is not a regular class and comes from a backport.
11:     """
12:
13:     def __str__(self):
14:         return '{0}: {1}'.format(self.name, self.value)
15:
16:     def __getitem__(self, index):
17:         return self.value[index]
18:
19:     def __len__(self):
20:         return len(self.value)
21:
22:     def __iter__(self):
23:         return iter(self.value)
24:
25:     def __eq__(self, other):
26:
27:         if not other or not hasattr(other, '__len__'):
28:             return False
29:
30:         if len(other) != len(self):
31:             return False
32:
33:         for index, value in enumerate(self):
34:             if not value == other[index]:
35:                 return False
36:
37:         return True
38:
39:     @classmethod
40:     def find_by_value(cls, value):
41:         for direction in cls:
42:             if value == direction:
43:                 return direction
44:
```



```
1: # -*- coding: utf-8 -*-
2: from engine.src.direction.direction import Direction
3:
4:
5: class EdgeDirection(Direction):
6:     """The 6 directions of a hexagon's edges with axial coordinates.
7:
8:     Each edge direction is a direction we can follow from the center of a
9:     hextile to a point on one of its edges.
10:
11:     Since each edge in a tile borders another tile, each edge direction
12:     also corresponds to a unit vector that we can follow from a given
13:     point in a hex axial coordinate system to get to another tile.
14:
15:     See more on axial coordinates here:
16:     http://www.redblobgames.com/grids/hexagons/#coordinates
17:     """
18:
19:     NORTH_WEST = (-1, 1, 0)
20:     NORTH_EAST = (0, 1, -1)
21:     WEST = (-1, 0, 1)
22:     EAST = (1, 0, -1)
23:     SOUTH_WEST = (0, -1, 1)
24:     SOUTH_EAST = (1, -1, 0)
25:
26:     def get_opposite_direction(self):
27:         """Get the direction of the opposite edge."""
28:
29:         coordinates = self.value
30:
31:         x = -coordinates[0]
32:         y = -coordinates[1]
33:         z = -(x + y)
34:
35:         return EdgeDirection.find_by_value((x, y, z))
```



```
1: # -*- coding: utf-8 -*-
2: from engine.src.direction.edge_direction import EdgeDirection
3: from engine.src.direction.vertex_direction import VertexDirection
4:
5:
6: class EdgeVertexMapping(object):
7:
8:     vertex_edge_mapping = {
9:         VertexDirection.TOP:
10:             (EdgeDirection.NORTH_WEST, EdgeDirection.NORTH_EAST),
11:         VertexDirection.TOP_RIGHT:
12:             (EdgeDirection.NORTH_EAST, EdgeDirection.EAST),
13:         VertexDirection.BOTTOM_RIGHT:
14:             (EdgeDirection.EAST, EdgeDirection.SOUTH_EAST),
15:         VertexDirection.BOTTOM:
16:             (EdgeDirection.SOUTH_EAST, EdgeDirection.SOUTH_WEST),
17:         VertexDirection.BOTTOM_LEFT:
18:             (EdgeDirection.SOUTH_WEST, EdgeDirection.WEST),
19:         VertexDirection.TOP_LEFT:
20:             (EdgeDirection.WEST, EdgeDirection.NORTH_WEST)
21:     }
22:
23:     edge_vertex_mapping = {
24:         EdgeDirection.NORTH_WEST:
25:             (VertexDirection.TOP_LEFT, VertexDirection.TOP),
26:         EdgeDirection.NORTH_EAST:
27:             (VertexDirection.TOP, VertexDirection.TOP_RIGHT),
28:         EdgeDirection.EAST:
29:             (VertexDirection.TOP_RIGHT, VertexDirection.BOTTOM_RIGHT),
30:         EdgeDirection.SOUTH_EAST:
31:             (VertexDirection.BOTTOM_RIGHT, VertexDirection.BOTTOM),
32:         EdgeDirection.SOUTH_WEST:
33:             (VertexDirection.BOTTOM, VertexDirection.BOTTOM_LEFT),
34:         EdgeDirection.WEST:
35:             (VertexDirection.BOTTOM_LEFT, VertexDirection.TOP_LEFT)
36:     }
37:
38:     @classmethod
39:     def get_edge_dirs_for_vertex_dir(cls, vertex_dir):
40:         """Returns directions of edges that share this vertex direction.
41:
42:         E.g. VertexDirection.TOP is the direction of the vertex that is an
43:         endpoint of both EdgeDirection.NORTH_WEST and EdgeDirection.NORTH_EAST.
44:
45:         Returns:
46:             tuple. A tuple of two directions, each of which has this vertex as
47:             an endpoint.
48:         """
49:
50:         return EdgeVertexMapping.vertex_edge_mapping[vertex_dir]
51:
52:     @classmethod
53:     def get_vertex_dirs_for_edge_dir(cls, edge_dir):
54:         """Get the vertex directions of endpoints of the given edge.
55:
56:         Returns:
57:             tuple. A tuple of 2 tuples, each of which is a value in
58:             VertexDirection that represents the endpoints of the given edge.
59:         """
60:
61:         return EdgeVertexMapping.edge_vertex_mapping[edge_dir]
```



```
1: # -*- coding: utf-8 -*-
2: from engine.src.direction.direction import Direction
3:
4:
5: class VertexDirection(Direction):
6:     """The 6 directions of a hexagon's vertices using cubic coordinates.
7:
8:     Each vertex direction is a direction we can follow from the center of a
9:     tile to one of its vertexes.
10:
11:     If we consider the hexagon a cube, the values correspond to the cubic
12:     (x, y, z) coordinates of the various directions.
13:
14:     See more on cubic coordinates here:
15:     http://www.redblobgames.com/grids/hexagons/#coordinates
16:     """
17:
18:     TOP = (1, 1, 0)
19:     TOP_RIGHT = (1, 0, 0)
20:     BOTTOM_RIGHT = (1, 0, 1)
21:     BOTTOM = (0, 0, 1)
22:     BOTTOM_LEFT = (0, 1, 1)
23:     TOP_LEFT = (0, 1, 0)
24:
25:     def get_opposite_direction(self):
26:         """Get the direction of the vertex opposite one of this direction."""
27:
28:         coordinates = self.value
29:
30:         def toggle(val):
31:             """Toggle val between 0 and 1."""
32:             return int(not bool(val))
33:
34:         x = toggle(coordinates[0])
35:         y = toggle(coordinates[1])
36:         z = toggle(coordinates[2])
37:
38:         return VertexDirection.find_by_value((x, y, z))
39:
40:     @classmethod
41:     def get_neighboring_vertex_dirs(cls, vertex_dir):
42:
43:         mapping = {
44:             VertexDirection.TOP:
45:                 (VertexDirection.TOP_LEFT, VertexDirection.TOP_RIGHT),
46:             VertexDirection.TOP_RIGHT:
47:                 (VertexDirection.TOP, VertexDirection.BOTTOM_RIGHT),
48:             VertexDirection.BOTTOM_RIGHT:
49:                 (VertexDirection.TOP_RIGHT, VertexDirection.BOTTOM),
50:             VertexDirection.BOTTOM:
51:                 (VertexDirection.BOTTOM_RIGHT, VertexDirection.BOTTOM_LEFT),
52:             VertexDirection.BOTTOM_LEFT:
53:                 (VertexDirection.BOTTOM, VertexDirection.TOP_LEFT),
54:             VertexDirection.TOP_LEFT:
55:                 (VertexDirection.BOTTOM_LEFT, VertexDirection.TOP),
56:         }
57:
58:         return mapping[vertex_dir]
59:
60:     @classmethod
61:     def pairs(cls):
62:         """Returns vertex pairs, each of which constitute an edge of a hex."""
63:
64:         return (
65:             (cls.TOP, cls.TOP_RIGHT),
66:             (cls.TOP_RIGHT, cls.BOTTOM_RIGHT),
67:             (cls.BOTTOM_RIGHT, cls.BOTTOM),
68:             (cls.BOTTOM, cls.BOTTOM_LEFT),
69:             (cls.BOTTOM_LEFT, cls.TOP_LEFT),
70:             (cls.TOP_LEFT, cls.TOP)
71:         )
```


./engine/src/edge.py

Thu Mar 26 18:19:20 2015

1

```
1: # -*- coding: utf-8 -*-  
2:  
3:  
4: class Edge(object):  
5:     pass
```



```

1: from engine.src.lib.utils import Utils
2:
3:
4: class UserMessageException(Exception):
5:     """
6:     A custom exception class that prints self.msg when cast to a string.
7:     """
8:     def __init__(self, msg):
9:         self.msg = msg
10:
11:     def __str__(self):
12:         return self.msg
13:
14:
15: class NotEnoughResourcesException(UserMessageException):
16:     """Raise when a trader lacks enough resources cards for a transaction.
17:
18:     E.g. when a player doesn't have enough resource cards to buy a structure,
19:     or when a bank runs out of resources.
20:
21:     Attributes:
22:         See Exception.
23:
24:     Args:
25:         trading_entity (TradingEntity): The entity that lacked resources.
26:
27:         resource_type (ResourceType or list of ResourceType): The type(s) of
28:         resource(s) the entity lacked.
29:     """
30:
31:     def __init__(self, trading_entity, resource_types):
32:
33:         resource_type_strs = map(
34:             lambda resource_type: str(resource_type),
35:             Utils.convert_to_list(resource_types)
36:         )
37:
38:         resource_type_str = ''
39:
40:         if len(resource_type_strs) == 1:
41:             resource_type_str = resource_type_strs[0]
42:         else:
43:             resource_type_str = ', '.join(resource_type_strs[:-1]) + \
44:                 ', or ' + resource_type_strs[-1]
45:
46:         self.msg = '{0} does not have enough {1} cards!'.format(
47:             trading_entity.__class__.__name__, resource_type_str)
48:
49:
50: class NotEnoughStructuresException(UserMessageException):
51:     """Raise when a player tries to build a structure despite having none left.
52:
53:     Args:
54:         player (Player): The player that tried to build a structure.
55:
56:         structure_name (str): The string name of structure the player attempted
57:         to build despite having run out.
58:     """
59:
60:     def __init__(self, player, structure_name):
61:         self.msg = '{0} does not have a {1} in stock.'.format(
62:             player.name, structure_name)
63:
64:
65: class NotEnoughDevelopmentCardsException(UserMessageException):
66:     """Raise when a player tries to buy a development card when none left."""

```

```

67:
68:     def __init__(self):
69:         self.msg = 'No development cards remaining.'
70:
71:
72: class InvalidBaseStructureException(UserMessageException):
73:     """Raise when one tries to build an invalid upgrade or extension structure.
74:
75:     Upgrade and extension structures need to be built off an appropriate base
76:     structure of a predetermined class. If the wrong class base structure is
77:     attempted, we should raise this error.
78:     """
79:
80:     def __init__(self, base_structure, augmenting_structure):
81:         augments = augmenting_structure.augments()
82:
83:         if augments is None:
84:             augments = 'an empty position'
85:
86:         self.msg = '{0} must replace {}, but tried to replace a {}'.format(
87:             augmenting_structure.name, augments, base_structure.name)
88:
89:
90: class BoardPositionOccupiedException(UserMessageException):
91:     """Raise when a player tries to build on a taken board position.
92:
93:     Players can not place structures on positions taken by other players.
94:     Players can not replace existing structures with non-augmenting structures.
95:     """
96:
97:     def __init__(self, position, structure, owning_player):
98:
99:         self.msg = 'Position {} already has a {} belonging to {}'.format(
100:             position, structure.name, owning_player.name)
101:
102:
103: class NoConfigValueDefinedException(UserMessageException):
104:
105:     def __init__(self, dot_notation_str):
106:
107:         self.msg = 'No config value defined for {}'.format(dot_notation_str)
108:
109:
110: class NoSuchVertexException(UserMessageException):
111:
112:     def __init__(self, tile, vertex_dir):
113:
114:         self.msg = 'Tile has no vertex: {}'.format(vertex_dir)
115:
116: class NoSuchEdgeException(UserMessageException):
117:
118:     def __init__(self, tile, edge_dir):
119:
120:         self.msg = 'Tile has no edge: {}'.format(edge_dir)
121:
122:
123: class InvalidStructurePlacementException(UserMessageException):
124:     """Raise when a player tries to place a structure somewhere they shouldn't.
125:
126:     E.g. no neighboring claimed roads, too close to another structure, etc.
127:     """
128:
129:     def __init__(self):
130:         self.msg = 'Not a valid position to place the structure.'

```



```

1: import pdb
2: from engine.src.config.config import Config
3: from engine.src.lib.utils import Utils
4: from engine.src.exceptions import *
5: from engine.src.player import Player
6: from engine.src.dice import Dice
7: from engine.src.trading.trade_offer import TradeOffer
8: from engine.src.input_manager import InputManager
9: from engine.src.board.game_board import GameBoard
10: from engine.src.resource_type import ResourceType
11: from engine.src.position_type import PositionType
12: from engine.src.structure.structure import Structure
13: from engine.src.calamity.robber import Robber
14: from engine.src.longest_road_search import LongestRoadSearch
15:
16: from imperative_parser.oracle import ORACLE
17:
18: class Game(object):
19:     """A game of Settlers of Catan."""
20:
21:     def __init__(self):
22:
23:         Config.init()
24:         ORACLE.set('game', self)
25:
26:         self.dice = Dice()
27:         self.board = GameBoard(Config.get('game.board.radius'))
28:         ORACLE.set('board', self.board)
29:
30:         # Place the robber on a fallow tile.
31:         self.robber = Robber()
32:         tile = self.board.get_tile_of_resource_type(ResourceType.FALLOW)
33:         tile.add_calamity(self.robber)
34:
35:         self.players = []
36:         self.input_manager = InputManager
37:
38:     def start(self):
39:         self.create_players()
40:         self.initial_settlement_and_road_placement()
41:         self.game_loop()
42:
43:     def game_loop(self):
44:
45:         max_point_count = 0
46:
47:         while max_point_count < Config.get('game.points_to_win'):
48:             for player in self.players:
49:                 ORACLE.set('player', player)
50:                 InputManager(self, player).cmdloop()
51:                 self.update_point_counts()
52:                 max_point_count = self.get_winning_player().get_total_points()
53:
54:         # Print out game over message.
55:         winner = self.get_winning_player()
56:         print 'Game over. {0} wins with {1} points!\n\'
57:             .format(winner.name, winner.get_total_points())
58:
59:     def create_players(self):
60:         """Create a new batch of players."""
61:
62:         self.players = []
63:         player_names = InputManager.get_player_names()
64:
65:         for player_name in player_names:
66:             self.players.append(Player(player_name))

```

```

67:
68:         ORACLE.set('players', self.players)
69:
70:     def place_structure(self, player, structure_name, must_border_claimed_edge=True,
71:                         struct_x=None, struct_y=None, struct_vertex_dir=None, free_t
o_build=False):
72:         """Place an edge or vertex structure.
73:
74:         Prompts for placement information and attempts to place on board. Does
75:         not do any exception handling.
76:         """
77:
78:         try:
79:
80:             structure = player.get_structure(structure_name)
81:
82:             if not free_to_build:
83:                 # Requesting structure, not further resources
84:                 trade_offer = TradeOffer(structure.cost, {})
85:                 obstructing_entity, obstructing_resource_type = \
86:                     trade_offer.validate(player, self.board.bank)
87:
88:                 if not obstructing_entity and not obstructing_resource_type:
89:                     trade_offer.execute(player, self.board.bank)
90:                 else:
91:                     raise NotEnoughResourcesException(obstructing_entity, obstructin
g_resource_type)
92:
93:             if structure.position_type == PositionType.EDGE:
94:                 prompt_func = InputManager.prompt_edge_placement
95:                 placement_func = self.board.place_edge_structure
96:             elif structure.position_type == PositionType.VERTEX:
97:                 prompt_func = InputManager.prompt_vertex_placement
98:                 placement_func = self.board.place_vertex_structure
99:
100:            x, y, struct_dir = prompt_func(self)
101:
102:            params = [x, y, struct_dir, structure, must_border_claimed_edge]
103:
104:            if struct_vertex_dir is not None:
105:                params.extend([struct_x, struct_y, struct_vertex_dir])
106:
107:            placement_func(*params)
108:
109:            player = structure.owning_player
110:
111:            # Allocate points
112:            if structure.augments():
113:                # TODO: conversions from camelcase to underscore
114:                points = structure.point_value - Config.get('game.structure.player_b
uilt.' + structure_name.lower()).point_value
115:            else:
116:                points = structure.point_value
117:
118:            player.points += points
119:
120:            return x, y, struct_dir, structure
121:
122:        except (NotEnoughStructuresException, NotEnoughResourcesException), e:
123:            raise
124:        except (BoardPositionOccupiedException, InvalidBaseStructureException,
125:                InvalidStructurePlacementException), e:
126:
127:            if not free_to_build:
128:                # If we bought the structure but didn't place it properly,
129:                # return the cost of the structure to the player.

```

```

130:         player.deposit_multiple_resources(structure.cost)
131:
132:         # And return the structure to their storage.
133:         player.restore_structure(structure_name)
134:
135:         # Raise the caught error so that callers of this method can handle
136:         # it in a custom fashion.
137:         raise
138:
139:     def place_init_structure(self, player, structure_name,
140:                             must_border_claimed_edge=False,
141:                             struct_x=None, struct_y=None,
142:                             struct_vertex_dir=None):
143:
144:         valid = False
145:
146:         while not valid:
147:             try:
148:                 free_to_build = True
149:
150:                 x, y, struct_dir, struct = self.place_structure(player, structure_name,
151:                         must_border_claimed_edge,
152:                         struct_x, struct_y, struct_vertex_dir, free_to_build)
153:
154:                 valid = True
155:             except (BoardPositionOccupiedException,
156:                     InvalidBaseStructureException,
157:                     InvalidStructurePlacementException), e:
158:                 player.restore_structure(structure_name)
159:                 InputManager.output(e)
160:
161:         return x, y, struct_dir
162:
163:     def initial_settlement_and_road_placement(self):
164:
165:         InputManager.announce_initial_structure_placement_stage()
166:
167:         for player in self.players:
168:             InputManager.announce_player_turn(player)
169:
170:             # Place settlement
171:             InputManager.announce_structure_placement(player, 'Settlement')
172:             x, y, vertex_dir = self.place_init_structure(player, 'Settlement')
173:
174:             # Place road
175:             InputManager.announce_structure_placement(player, 'Road')
176:             self.place_init_structure(player, 'Road', False, x, y, vertex_dir)
177:
178:         distributions = Utils.nested_dict()
179:
180:         for player in list(reversed(self.players)):
181:             InputManager.announce_player_turn(player)
182:
183:             # Place settlement
184:             InputManager.announce_structure_placement(player, 'Settlement')
185:             x, y, vertex_dir = self.place_init_structure(player, 'Settlement')
186:
187:             # Place road
188:             InputManager.announce_structure_placement(player, 'Road')
189:             self.place_init_structure(player, 'Road', False, x, y, vertex_dir)
190:
191:         neighboring_tiles = filter(
192:             bool, self.board.get_adjacent_tiles_to_vertex(x, y, vertex_dir))
193:

```

```

194:
195:         # Give initial resource cards
196:         resource_types = filter(
197:             lambda resource_type: resource_type != ResourceType.FALLOW,
198:             map(lambda tile: tile.resource_type, neighboring_tiles)
199:         )
200:
201:         for resource_type in resource_types:
202:
203:             if not distributions[player][resource_type]:
204:                 distributions[player][resource_type] = 0
205:
206:             distributions[player][resource_type] += \
207:                 Config.get('game.structure.player_built.settlement.base_yield')
208:
209:         self.board.distribute_resources(distributions)
210:         InputManager.announce_resource_distributions(distributions)
211:
212:     def roll_dice(self, value=None):
213:
214:         roll_value = self.dice.roll()
215:         InputManager.announce_roll_value(roll_value)
216:         ORACLE.set('dice_value', roll_value)
217:
218:         # If a calamity value, handle calamity
219:         distributions = self.board.distribute_resources_for_roll(roll_value)
220:
221:         InputManager.announce_resource_distributions(distributions)
222:
223:     def get_winning_player(self):
224:         """Get the player who is winning this game of Settlers of Catan."""
225:
226:         return max(self.players, key=lambda player: player.points)
227:
228:     def update_point_counts(self):
229:
230:         for player in self.players:
231:             player.special_points = 0
232:
233:         player_with_largest_army = max(self.players, key=lambda player: player.knights)
234:
235:         # TODO: Move thresholds to config
236:         if player_with_largest_army.knights >= 3:
237:             print 'Largest army given to: {}'.format(player_with_largest_army)
238:             player_with_largest_army.special_points += 2
239:
240:         player_road_len_dict = LongestRoadSearch(self.board).execute()
241:
242:         for player, road_len in player_road_len_dict.iteritems():
243:             player.longest_road_length = road_len
244:
245:         player_with_longest_road = max(player_road_len_dict)
246:
247:         if player_with_longest_road.longest_road_length >= 5:
248:             print 'Longest road given to: {}'.format(player_with_longest_road)
249:             player_with_longest_road.special_points += 2

```

```

1: import cmd
2: import sys
3: import pdb
4:
5: from engine.src.config.config import Config
6: from engine.src.direction.vertex_direction import VertexDirection
7: from engine.src.direction.edge_direction import EdgeDirection
8: from engine.src.resource_type import ResourceType
9: from engine.src.vertex import Vertex
10: from engine.src.edge import Edge
11: from engine.src.exceptions import *
12: from engine.src.trading.trade_offer import TradeOffer
13: from engine.src.structure.structure import Structure
14:
15:
16: class InputManager(cmd.Cmd):
17:     """Class managing input for a given player's turn. See docs for cmd.Cmd.
18:
19:     Args:
20:         game (Game): The game being played.
21:         player (Player): Current player.
22:
23:     Note that method docstrings are displayed to the user when they enter help.
24:     Implementation documentation should thus be given below the usual docstring.
25:     TODO: Commands do not support cancellation part way through.
26:     """
27:     def __init__(self, game, player):
28:
29:         cmd.Cmd.__init__(self)
30:
31:         self.game = game
32:         self.player = player
33:         self.prompt = '> {0}: '.format(self.player.name)
34:
35:         self.has_rolled = False
36:         self.has_played_card = False
37:
38:         self.structure_names = Utils.pluck(Config.get('game.structure.player_built'),
, 'name')
39:
40:     def emptyline(self, line=None):
41:         """Override default emptyline behavior, which repeats last command."""
42:         if line is None:
43:             return
44:         self.default(line)
45:
46:     def default(self, line):
47:         """Print menu of commands when unrecognized command given."""
48:
49:         print 'Unrecognized command <{0}> given.'.format(line)
50:         self.do_help(None)
51:
52:     def preloop(self):
53:         """Announce start of player turn."""
54:
55:         msg = "{0}'s turn: ".format(self.player.name)
56:         InputManager.output(msg)
57:
58:     def postloop(self):
59:         """Announce end of player turn."""
60:
61:         msg = "End of {0}'s turn.".format(self.player.name)
62:         InputManager.output(msg)
63:
64:     def do_debug(self, line):
65:         pdb.set_trace()

```

```

66:
67:     def do_roll(self, value):
68:         """Roll the dice."""
69:
70:         self.game.roll_dice(value)
71:         self.has_rolled = True
72:
73:     # TODO: Move core logic to game.
74:     def do_trade_player(self, line):
75:         """Trade resources with other players."""
76:
77:         if not self.has_rolled:
78:             print 'You must roll before you can trade.'
79:             return
80:         else:
81:
82:             # Get list of requested resources
83:             msg = "Please enter a comma separated list of the number(s) " + \
84:                 "of the resource(s) you would like to offer."
85:
86:             # offered_resources => resource_type => count
87:             offered_resources = InputManager.prompt_select_list_subset(
88:                 msg, ResourceType.get_arable_types(),
89:                 self.player.validate_resources
90:             )
91:
92:             # Take csv list of offered resources
93:             msg = "Please enter a comma separated list of the number(s) " + \
94:                 "of the resource(s) you would like to receive."
95:
96:             # requested_resources => resource_type => count
97:             requested_resources = InputManager.prompt_select_list_subset(
98:                 msg, ResourceType.get_arable_types()
99:             )
100:
101:             # Create a trade offer
102:             trade_offer = TradeOffer(offered_resources, requested_resources)
103:
104:             # Get player who will give requested resources and receive
105:             # offered resources.
106:             msg = "Please enter the number (e.g. '1') of the player " + \
107:                 "you would like to trade with."
108:
109:             tradeable_players = filter(lambda player: player != self.player,
110:                                     self.game.players)
111:
112:             if not tradeable_players:
113:                 msg = 'No players to trade with.'
114:                 InputManager.output(msg)
115:                 return
116:
117:             other_player = InputManager.prompt_select_list_value(
118:                 msg, map(lambda player: player.name, tradeable_players),
119:                 tradeable_players
120:             )
121:
122:             try:
123:                 other_player.trade(self.player, trade_offer)
124:
125:                 distributions = {
126:                     self.player: requested_resources,
127:                     other_player: offered_resources
128:                 }
129:
130:                 InputManager.announce_trade_completed(trade_offer)
131:                 # TODO: Specify explicit possible exceptions.
132:                 except Exception as e:

```

```

132:         InputManager.output(e)
133:
134:     def do_trade_bank(self, line):
135:         """Trade resources with the bank"""
136:
137:         if not self.has_rolled:
138:             print 'You must roll before you can trade.'
139:             return
140:         else:
141:             # Get list of requested resources
142:             msg_offer = "Please enter the number of the resource you want to offer."
+ \
143:             "The bank buys 4 of a given resource, and returns 1 of any other res
ource."
144:
145:             offered_resource_type = InputManager.prompt_select_list_value(
146:                 msg_offer, ResourceType.get_arable_types()
147:             )
148:
149:             msg_request = "Please enter the number of the resource you want to reque
st."
150:
151:             requested_resource_type = InputManager.prompt_select_list_value(
152:                 msg_request, ResourceType.get_arable_types()
153:             )
154:
155:             offered_resources = {offered_resource_type: 4}
156:             requested_resources = {requested_resource_type: 1}
157:
158:             trade_offer = TradeOffer(offered_resources, requested_resources)
159:
160:             try:
161:                 self.game.board.bank.trade(self.player, trade_offer)
162:                 InputManager.announce_trade_completed(trade_offer)
163:
164:                 # TODO: Specify explicit possible exceptions.
165:             except Exception as e:
166:                 InputManager.output(e)
167:
168:         # TODO
169:         # TODO: long term. Refactor to be compatible w/ any trade intermediary.
170:
171:     def do_trade_harbor(self, line):
172:         """Trade resources with a harbor."""
173:         print('not yet implemented')
174:
175:     def do_build(self, line):
176:         """Build structures, including settlements, cities, and roads."""
177:
178:         if not self.has_rolled:
179:             print 'You must roll before you can build.'
180:             return
181:
182:         try:
183:             msg = "Please enter the number (e.g. '1') of the structure " + \
184:                 "you would like to build."
185:
186:             structure_name = InputManager.prompt_select_list_value(
187:                 msg, self.structure_names)
188:
189:             self.game.place_structure(self.player, structure_name)
190:
191:             self.game.update_point_counts()
192:
193:         except (NotEnoughStructuresException, NotEnoughResourcesException,
194:             BoardPositionOccupiedException, InvalidBaseStructureException,

```

```

195:             InvalidStructurePlacementException), e:
196:                 InputManager.output(e)
197:
198:         # TODO: Enforce can't play card bought during same turn.
199:     def do_buy_card(self, line):
200:         """Buy a development card."""
201:
202:         if not self.has_rolled:
203:             msg = 'You must roll before you can buy a development card.'
204:             InputManager.output(msg)
205:         elif self.has_played_card:
206:             msg = 'You may only play one card per turn.'
207:             InputManager.output(msg)
208:         else:
209:
210:             try:
211:                 dev_card = self.game.board.bank.buy_development_card(self.player)
212:                 dev_card.draw_card()
213:
214:                 success_msg = 'You received a {0}!'.format(str(dev_card))
215:
216:                 InputManager.input_default(success_msg, None, False)
217:
218:             except (NotEnoughDevelopmentCardsException, NotEnoughResourcesException)
as e:
219:                 InputManager.output(e)
220:
221:     def do_play_card(self, line):
222:         """Play a development card."""
223:
224:         if self.has_played_card:
225:             msg = 'You may only play one card per turn.'
226:             InputManager.output(msg)
227:         else:
228:
229:             msg = "Please enter the number (e.g. '1') of the development " + \
230:                 "card you would like to play."
231:
232:             dev_card = InputManager.prompt_select_list_value(
233:                 msg,
234:                 map(lambda card: card.name, self.player.get_unplayed_development_car
ds()),
235:                 self.player.get_unplayed_development_cards()
236:             )
237:
238:             if not dev_card:
239:                 InputManager.input_default(
240:                     'Player has no development cards to choose from',
241:                     None, False)
242:                 return
243:
244:             try:
245:                 dev_card.play_card()
246:                 self.game.update_point_counts()
247:
248:             # TODO: Make clear which exceptions can be caught.
249:             except Exception as e:
250:                 InputManager.output(e)
251:
252:         # TODO: Improve.
253:     def do_print_board(self, line):
254:         """View the board."""
255:
256:         for tile in self.game.board.iter_tiles():
257:             print tile
258:

```



```

259:     def do_view_points(self, line):
260:         """View points per player (not including other players' hidden points)."""
261:
262:         msg = 'Player Point Counts:\n'
263:
264:         for player in self.game.players:
265:             points = player.get_total_points() if player == self.player \
266:                 else player.get_visible_points()
267:             msg += '{:}\t{}'.format(player, points)
268:
269:         InputManager.output(msg)
270:
271:     def do_view_resources(self, line):
272:         """View your resource cards."""
273:
274:         msg = '\n' + '\n'.join(map(
275:             lambda resource_type: '{}:\t{}'.format(resource_type, self.player.resour
ces[resource_type]),
276:             self.player.resources
277:         ))
278:
279:         InputManager.output(msg)
280:
281:     # TODO
282:     def do_view_structures(self, line):
283:         """View your vertex and edge structures."""
284:
285:         edge_structures = []
286:         vertex_structures = []
287:
288:         for x, y in self.game.board.iter_tile_coords():
289:             tile = self.game.board.get_tile_with_coords(x, y)
290:
291:             if not tile:
292:                 continue
293:
294:             for edge_dir in EdgeDirection:
295:                 edge_val = tile.get_edge(edge_dir)
296:
297:                 if isinstance(edge_val, Structure) and \
298:                     edge_val.owning_player == self.player:
299:
300:                     edge_structures.append( (tile, edge_dir, edge_val) )
301:
302:             for vertex_dir in VertexDirection:
303:                 vertex_val = tile.get_vertex(vertex_dir)
304:
305:                 if isinstance(vertex_val, Structure) and \
306:                     vertex_val.owning_player == self.player:
307:                     vertex_structures.append( (tile, vertex_dir, vertex_val) )
308:
309:         structures = []
310:         tups_to_print = []
311:
312:         for s in edge_structures:
313:             if s[2] not in structures:
314:                 structures.append(s[2])
315:                 tups_to_print.append(s)
316:
317:         for s in vertex_structures:
318:             if s[2] not in structures:
319:                 structures.append(s[2])
320:                 tups_to_print.append(s)
321:
322:         msg = '\n' + '\n'.join(map(lambda tup: 'Tile: {} \tDirection: {} \tStructure:
{}'.format(

```

```

323:             tup[0], tup[1], tup[2].name), tups_to_print))
324:
325:         InputManager.output(msg)
326:
327:     def do_end_turn(self, line):
328:         """End your current turn."""
329:
330:         if not self.has_rolled:
331:             print 'You must roll before you can end your turn.'
332:         else:
333:             return True
334:
335:     def do_quit(self, line):
336:         """Quit the game for all players."""
337:         print '\nYou quit the game.'
338:         sys.exit(0)
339:
340:     # Testing Methods
341:     def do_aybabbu(self, count):
342:         """All your base are belong to us."""
343:
344:         if not count:
345:             count = 100
346:         else:
347:             count = int(count)
348:
349:         for resource_type in ResourceType.get_arable_types():
350:             self.player.deposit_resources(resource_type, count)
351:
352:     @staticmethod
353:     def output(msg):
354:         """Outputs the given message."""
355:         InputManager.input_default(msg, None, False)
356:
357:     @staticmethod
358:     def input_default(msg, default=None, read_result=True):
359:         """Asks for user data using the format specified below.
360:
361:         Returns:
362:             str. string entered by the user, or default if nothing was entered.
363:         """
364:
365:         prompt = '> {}'.format(str(msg))
366:
367:         if default:
368:             prompt += " (or press enter to use default {}): ".format(default)
369:
370:         if read_result:
371:             prompt += '\n< '
372:             result = raw_input(prompt)
373:             # TODO: only return default if default flag true
374:             return result if result else default
375:         else:
376:             print prompt
377:
378:     @staticmethod
379:     def get_player_names():
380:         """Prompts for and takes in player names.
381:
382:         Returns:
383:             list. Of player name strings.
384:         """
385:         player_names = []
386:         num_players = 0
387:
388:         while num_players <= 0:

```

```

389:         try:
390:             num_players = int(
391:                 InputManager.input_default(
392:                     'Enter number of players',
393:                     Config.get('game.player_count')
394:                 )
395:             )
396:
397:             if num_players <= 0:
398:                 raise ValueError
399:
400:         except ValueError:
401:             msg = 'Invalid number of players. Number must be an integer' + \
402:                 ' greater than zero.'
403:             InputManager.output(msg)
404:
405:         # Shift range by 1 so prompts starting with player 1, not player 0
406:         for i in range(1, num_players + 1):
407:             msg = "Specify player {0}'s name".format(i)
408:             default = 'p{0}'.format(i)
409:             player_name = InputManager.input_default(msg, default)
410:             player_names.append(player_name)
411:
412:         return player_names
413:
414:     @staticmethod
415:     def prompt_select_player(game, players=None):
416:
417:         if players is None:
418:             players = game.players
419:
420:         msg = "Please enter the number (e.g. '1') of the player" + \
421:             "you would like to choose."
422:
423:         return InputManager.prompt_select_list_value(msg, players)
424:
425:     @staticmethod
426:     def prompt_tile_coordinates(game):
427:
428:         x, y = None, None
429:
430:         valid_coords = False
431:
432:         while not valid_coords:
433:             try:
434:                 x = int(InputManager.input_default(
435:                     'Please specify a tile x coordinate:', None))
436:
437:                 y = int(InputManager.input_default(
438:                     'Please specify a tile y coordinate:', None))
439:
440:                 valid_coords = game.board.valid_tile_coords(x, y)
441:
442:                 if not valid_coords:
443:                     raise ValueError
444:             except Exception:
445:                 error_msg = "Invalid coordinates. Please try again."
446:                 InputManager.output(error_msg)
447:
448:         return x, y
449:
450:     @staticmethod
451:     def prompt_select_list_value(prompt_msg, display_list, value_list=None):
452:         """Select and return a list element.
453:
454:         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:

```

```

521:         resource_count_dict = Utils.convert_list_to_count_dict(map(
522:             lambda index: allowed_values_lst[int(index) - 1],
523:             index_list
524:         ))
525:
526:         valid = validate_func(resource_count_dict) \
527:             if validate_func is not None else True
528:
529:     except (IndexError, ValueError):
530:         msg = "Invalid number given. All numbers must be " + \
531:             "between 1 and {0}.".format(len(allowed_values_lst))
532:         InputManager.output(msg)
533:     except NotEnoughResourcesException as n:
534:         InputManager.output(n)
535:
536:     return resource_count_dict
537:
538: @staticmethod
539: def prompt_select_resource_type():
540:
541:     msg = "Please enter the number (e.g. '1') of the resource type" + \
542:         "you would like to choose."
543:
544:     return InputManager.prompt_select_list_value(msg, list(ResourceType))
545:
546: @staticmethod
547: def prompt_vertex_direction():
548:
549:     msg = "Please enter the number (e.g. '1') of the direction " + \
550:         "from the center of the tile to the vertex you would " + \
551:         "like to place a structure on."
552:
553:     return InputManager.prompt_select_list_value(msg, list(VertexDirection))
554:
555: @staticmethod
556: def prompt_edge_direction():
557:
558:     msg = "Please enter the number (e.g. '1') of the direction " + \
559:         "from the center of the tile to the edge you would " + \
560:         "like to place a structure on."
561:
562:     return InputManager.prompt_select_list_value(msg, list(EdgeDirection))
563:
564: @staticmethod
565: def prompt_vertex_placement(game):
566:
567:     x, y = InputManager.prompt_tile_coordinates(game)
568:
569:     vertex_dir = InputManager.prompt_vertex_direction()
570:
571:     return x, y, vertex_dir
572:
573: @staticmethod
574: def prompt_edge_placement(game):
575:
576:     x, y = InputManager.prompt_tile_coordinates(game)
577:
578:     edge_dir = InputManager.prompt_edge_direction()
579:
580:     return x, y, edge_dir
581:
582: # TODO: Roll announce methods into single method? Or programatically set.
583:
584: @staticmethod
585: def announce_roll_value(roll_value):
586:

```

```

587:     prompt = 'Player rolled a {0}'.format(roll_value)
588:     InputManager.output(prompt)
589:
590: @staticmethod
591: def announce_initial_structure_placement_stage():
592:
593:     prompt = 'Beginning initial structure placement stage.'
594:     InputManager.output(prompt)
595:
596: @staticmethod
597: def announce_player_turn(player):
598:
599:     prompt = "Beginning {0}'s turn.".format(player.name)
600:     InputManager.output(prompt)
601:
602: @staticmethod
603: def announce_structure_placement(player, structure_name):
604:
605:     prompt = "{0}, select where you would like to place your {1}".format(
606:         player.name, structure_name
607:     )
608:     InputManager.output(prompt)
609:
610: @staticmethod
611: def announce_development_card_played(player, development_card):
612:
613:     prompt = "{0} played a development card: {1}".format(
614:         player.name, str(development_card))
615:     InputManager.output(prompt)
616:
617: @staticmethod
618: def announce_resource_distributions(distributions):
619:
620:     msg = 'Distributing resources.'
621:     InputManager.output(msg)
622:
623:     for player in distributions:
624:         for resource_type in distributions[player]:
625:             count = distributions[player][resource_type]
626:
627:             if count:
628:                 msg = '{0} received {1} {2} cards.'.format(
629:                     player.name, count, resource_type)
630:                 InputManager.output(msg)
631:
632: @staticmethod
633: def announce_trade_completed(trade_offer):
634:     requested_resources = trade_offer.requested_resources
635:     offered_resources = trade_offer.offered_resources
636:
637:     def generate_resources_readable_str(resources):
638:         return ", ".join(map(
639:             lambda res: str(resources[res]) + " " + str(res) + "(s)",
640:             (res for res in resources if resources[res] != 0)
641:         ))
642:
643:     msg = "Trade completed. You bought " + \
644:         generate_resources_readable_str(requested_resources) + " and sold " + \
645:         generate_resources_readable_str(offered_resources) + "."
646:
647:     InputManager.output(msg)

```



```

1: # -*- coding: utf-8 -*-
2: import collections
3: from types import MethodType
4:
5: def noop(cls, *args, **kwargs):
6:     pass
7:
8: class Utils(object):
9:     """A general utility class."""
10:
11:     @classmethod
12:     def init_from_dict(cls, obj, dct):
13:
14:         for key, val in dct.iteritems():
15:             if Utils.is_function(val):
16:                 setattr(obj, key, MethodType(val, obj, obj.__class__))
17:             else:
18:                 setattr(obj, key, val)
19:
20:     @classmethod
21:     def pluck(cls, dct, prop, do_filter=False):
22:         """Gets a list of values for the given property.
23:
24:         Assumes the dct has key-value pairs where values are also dcts. Gets
25:         a list of values for the given property by taking them off each such
26:         value dct.
27:         """
28:
29:         lst = []
30:
31:         try:
32:             lst = map(lambda key: dct[key][prop], dct)
33:
34:             if do_filter:
35:                 lst = filter(lambda value: value is not None, lst)
36:
37:         except KeyError:
38:             lst = []
39:
40:         return lst
41:
42:     @classmethod
43:     def remove_duplicates(cls, lst):
44:
45:         result = []
46:
47:         for e in lst:
48:             if e not in result:
49:                 result.append(e)
50:
51:         return result
52:
53:     @classmethod
54:     def is_function(cls, func):
55:         return hasattr(func, '__call__')
56:
57:     @classmethod
58:     def is_list(cls, lst):
59:         return hasattr(lst, "__iter__")
60:
61:     @classmethod
62:     def noop(cls, *args, **kwargs):
63:         pass
64:
65:     @classmethod
66:     def flatten(cls, lst):

```

```

67:         """Flattens a 2D list of lists."""
68:
69:         return [nested_elem for elem in lst for nested_elem in elem]
70:
71:     @classmethod
72:     def nested_dict(cls):
73:         """A nested default dictionary.
74:
75:         Dictionaries in Python can become cumbersome if you constantly have to
76:         check if a key exists in a dictionary before proceeding. Using this as
77:         a dict definition allows the user to define arbitrarily nested values
78:         in the dictionary. Undefined nested values will return a defaultdict
79:         that, when cast to a boolean, will return False.
80:
81:         Usage:
82:             my_dict = Utils.nested_dict()
83:             my_dict[k1][k2][k3] = value
84:
85:         Taken from:
86:             http://stackoverflow.com/questions/16724788/how-can-i-get-python-to-automatically-create-missing-key-value-pairs-in-a-dictio
87:         """
88:         return collections.defaultdict(cls.nested_dict)
89:
90:     @classmethod
91:     def convert_list_to_count_dict(cls, lst):
92:
93:         dct = {}
94:
95:         for val in lst:
96:             if val in dct:
97:                 dct[val] += 1
98:             else:
99:                 dct[val] = 1
100:
101:         return dct
102:
103:     @classmethod
104:     def convert_to_list(cls, e):
105:         """Convert to a list if not already a list."""
106:         return [e] if not Utils.is_list(e) else e
107:
108:     @classmethod
109:     def dict_to_list(cls, dct):
110:         """Convert a counter-like dict to a list."""
111:         return Utils.flatten(map(lambda k: [k] * dct[k], dct))
112:
113:     @classmethod
114:     def convert_format(cls, str):
115:         return str.replace('-', '_')
116:

```



```

1: import pdb
2: from engine.src.lib.utils import Utils
3: from engine.src.direction.edge_direction import EdgeDirection
4: from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping
5: from engine.src.structure.structure import Structure
6: from engine.src.tile.hex_tile import HexTile
7:
8:
9: global vertices
10: global edges
11:
12:
13: def reset metas():
14:     global vertices
15:     global edges
16:     vertices = Utils.nested_dict()
17:     edges = Utils.nested_dict()
18:
19:
20: def find_edge_meta(board, x, y, edge_dir):
21:     edge = edges[x][y][edge_dir]
22:
23:     if not edge:
24:         tile = board.get_tile_with_coords(x, y)
25:         if tile:
26:             edge = EdgeMeta(board, x, y, edge_dir)
27:         else:
28:             edge = None
29:
30:     return edge
31:
32:
33: def find_vertex_meta(board, x, y, vertex_dir):
34:     vertex = vertices[x][y][vertex_dir]
35:
36:     if not vertex:
37:         tile = board.get_tile_with_coords(x, y)
38:         if tile:
39:             vertex = VertexMeta(board, x, y, vertex_dir)
40:         else:
41:             vertex = None
42:
43:     return vertex
44:
45:
46: class VertexMeta(object):
47:
48:     def __init__(self, board, x, y, vertex_dir):
49:
50:         vertices[x][y][vertex_dir] = self
51:
52:         self.board = board
53:
54:         self.x = x
55:         self.y = y
56:         self.tile = self.board.get_tile_with_coords(self.x, self.y)
57:
58:         self.vertex_dir = vertex_dir
59:
60:         self.neighbors = []
61:
62:         self.neighbors = self.find_neighbor_equivalents()
63:
64:     def find_neighbor_equivalents(self):
65:
66:         neighbors = []

```

```

67:
68:     # Get the two edges of the found tile that have as an endpoint
69:     # a vertex of the given vertex direction.
70:     vertex_adj_edge_dirs = EdgeVertexMapping.get_edge_dirs_for_vertex_dir(
71:         self.vertex_dir)
72:
73:     for vertex_adj_edge_dir in vertex_adj_edge_dirs:
74:         neighbor_x = self.tile.x + vertex_adj_edge_dir[0]
75:         neighbor_y = self.tile.y + vertex_adj_edge_dir[1]
76:         neighbor_tile = self.board.get_neighboring_tile(self.tile, vertex_adj_ed
77: ge_dir)
78:
79:         # Edge tiles may not have neighboring tiles in the given direction.
80:         if neighbor_tile:
81:             neighbor_vertex_dir = HexTile.get_equivalent_vertex_dir(
82:                 self.vertex_dir, vertex_adj_edge_dir)
83:             neighbor = find_vertex_meta(self.board, neighbor_x, neighbor_y, neig
84: hbor_vertex_dir)
85:             neighbors.append(neighbor)
86:
87:     return neighbors
88:
89:     def __str__(self):
90:         return '({}, {}) {}'.format(self.x, self.y, self.vertex_dir)
91:
92:     def __eq__(self, other):
93:
94:         matches = self.x == other.x and \
95:             self.y == other.y and \
96:             self.vertex_dir == other.vertex_dir
97:
98:         for neighbor in self.neighbors:
99:             matches = matches or \
100:                 neighbor.x == other.x and \
101:                 neighbor.y == other.y and \
102:                 neighbor.vertex_dir == other.vertex_dir
103:
104:         return matches
105:
106:
107: class EdgeMeta(object):
108:
109:     def __init__(self, board, x, y, edge_dir):
110:
111:         edges[x][y][edge_dir] = self
112:
113:         self.board = board
114:
115:         self.x = x
116:         self.y = y
117:         self.tile = self.board.get_tile_with_coords(self.x, self.y)
118:
119:         self.edge_dir = edge_dir
120:         self.edge_val = self.tile.get_edge(self.edge_dir)
121:
122:         # Neighbor equivalent edge meta of same edge.
123:         self.neighbor_x = self.tile.x + self.edge_dir[0]
124:         self.neighbor_y = self.tile.y + self.edge_dir[1]
125:         self.neighbor_edge_dir = self.edge_dir.get_opposite_direction()
126:
127:         edges[self.neighbor_x][self.neighbor_y][self.neighbor_edge_dir] = self
128:
129:     def __str__(self):
130:         return '({}, {}) {}'.format(self.x, self.y, self.edge_dir)

```

```

131:
132:     def __repr__(self):
133:         return '({}, {}) {}'.format(self.x, self.y, self.edge_dir)
134:
135:     def __eq__(self, other):
136:
137:         matches_this = self.x == other.x and \
138:             self.y == other.y and \
139:             self.edge_dir == other.edge_dir
140:
141:         matches_neighbor = self.neighbor_x == other.x and \
142:             self.neighbor_y == other.y and \
143:             self.neighbor_edge_dir == other.edge_dir
144:
145:         return matches_this or matches_neighbor
146:
147: class LongestRoadSearch(object):
148:
149:     def __init__(self, board):
150:         self.board = board
151:
152:     def execute(self):
153:         reset metas()
154:
155:         player_claimed_edges_dict = self.find_per_player_claimed_edges()
156:         player_road_len_dict = self.find_per_player_max_road_lengths(player_claimed_
edges_dict)
157:
158:         return player_road_len_dict
159:
160:     def find_per_player_claimed_edges(self):
161:
162:         player_claimed_edges_dict = Utils.nested_dict()
163:         checked_edges = Utils.nested_dict()
164:
165:         for x, y in self.board.iter_tile_coords():
166:             tile = self.board.get_tile_with_coords(x, y)
167:
168:             if not tile:
169:                 continue
170:
171:             for edge_dir in EdgeDirection:
172:                 if not checked_edges[x][y][edge_dir]:
173:                     self.add_edge_to_dicts(x, y, edge_dir, player_claimed_edges_dict
, checked_edges)
174:
175:             return player_claimed_edges_dict
176:
177:     def add_edge_to_dicts(self, x, y, edge_dir, player_claimed_edges_dict, checked_e
dges):
178:
179:         edge_meta = find_edge_meta(self.board, x, y, edge_dir)
180:
181:         if not edge_meta:
182:             checked_edges[x][y][edge_dir] = True
183:             return
184:
185:         checked_edges[edge_meta.x][edge_meta.y][edge_meta.edge_dir] = True
186:         checked_edges[edge_meta.neighbor_x][edge_meta.neighbor_y][edge_meta.neighbor
_edge_dir] = True
187:
188:         if isinstance(edge_meta.edge_val, Structure):
189:             player = edge_meta.edge_val.owning_player
190:
191:             if not player_claimed_edges_dict[player]:
192:                 player_claimed_edges_dict[player] = []

```

```

193:
194:         player_claimed_edges_dict[player].append(edge_meta)
195:
196:     def find_per_player_max_road_lengths(self, player_claimed_edges_dict):
197:
198:         player_road_len_dict = {}
199:
200:         for player, player_claimed_edges in player_claimed_edges_dict.iteritems():
201:             player_road_len_dict[player] = self.find_max_road_len(player_claimed_edg
es)
202:
203:         return player_road_len_dict
204:
205:     def find_max_road_len(self, player_claimed_edges):
206:         """
207:         Args:
208:             player_claimed_edges (list): List of EdgeMetas.
209:         """
210:
211:         max_road_len = 0
212:
213:         for edge_meta in player_claimed_edges:
214:             edge_dir = edge_meta.edge_dir
215:
216:             vertex_dirs = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)
217:
218:             remaining_edges = [e for e in player_claimed_edges if e != edge_meta]
219:
220:             start_vertex = find_vertex_meta(self.board, edge_meta.x, edge_meta.y, ve
rtex_dirs[0])
221:             end_vertex = find_vertex_meta(self.board, edge_meta.x, edge_meta.y, vert
ex_dirs[1])
222:
223:             road_len = 1 + self.find_max_path_len(remaining_edges, end_vertex, edge_
meta) \
224:                 + self.find_max_path_len(remaining_edges, start_vertex, edge_
e_meta)
225:
226:             if road_len > max_road_len:
227:                 max_road_len = road_len
228:
229:         return max_road_len
230:
231:     def find_max_path_len(self, remaining_edges, end_vertex, edge_meta):
232:
233:         neighbor_edge_metas = map(
234:             lambda edge_tuple: find_edge_meta(self.board, *edge_tuple),
235:             self.board.get_adjacent_edges(edge_meta.x, edge_meta.y, end_vertex.verte
x_dir, False)
236:         )
237:
238:         claimed_neighbors = [i for i in neighbor_edge_metas if i in remaining_edges]
239:
240:         if claimed_neighbors:
241:             max_path_len = 0
242:
243:             for claimed_neighbor in claimed_neighbors:
244:                 remaining_edge_metas = [x for x in remaining_edges if (x != claimed_
neighbor and x != edge_meta)]
245:
246:                 vertices = EdgeVertexMapping.get_vertex_dirs_for_edge_dir(claimed_ne
ighbor.edge_dir)
247:
248:                 vertex_metas = map(
249:                     lambda vertex_dir: find_vertex_meta(self.board, claimed_neighbor
.x, claimed_neighbor.y, vertex_dir),

```



```
250:         vertices
251:     )
252:
253:     next_end_vertex = next(d for d in vertex metas if d != end_vertex)
254:
255:     path_len = 1 + self.find_max_path_len(remaining_edge_metas, next_end
_vertex, claimed_neighbor)
256:
257:     if path_len > max_path_len:
258:         max_path_len = path_len
259:
260:     return max_path_len
261: else:
262:     return 0
```



```
1: # -*- coding: utf-8 -*-
2: from engine.src.lib.utils import Utils
3: from engine.src.config.config import Config
4: from engine.src.structure.structure import Structure
5: from engine.src.trading.trading_entity import TradingEntity
6: from engine.src.exceptions import NotEnoughStructuresException
7:
8:
9: class Player(TradingEntity):
10:     """A player in a game of Settlers of Catan.
11:
12:     Attributes:
13:         resources (dict): See TradingEntity.
14:
15:         name (str): This player's name.
16:
17:     Args:
18:         name (str): Name to assign a new player.
19:     """
20:
21:     def __init__(self, name):
22:
23:         super(Player, self).__init__()
24:
25:         self.name = name
26:
27:         self.development_cards = []
28:
29:         self.points = 0
30:         self.hidden_points = 0
31:         self.special_points = 0
32:
33:         self.knights = 0
34:         self.longest_road_length = 0
35:
36:         self.remaining_structure_counts = {}
37:         self.init_structure_counts()
38:
39:     def __hash__(self):
40:         return hash(self.name)
41:
42:     def __eq__(self, other):
43:         return self.name == other.name
44:
45:     def __str__(self):
46:         return self.name
47:
48:     def init_structure_counts(self):
49:
50:         self.remaining_structure_counts = {}
51:
52:         for structure in Config.get('game.structure.player_built').values():
53:             self.remaining_structure_counts[structure['name']] = structure['count']
54:
55:     def get_total_points(self):
56:         return self.points + self.hidden_points + self.special_points
57:
58:     def get_unplayed_development_cards(self):
59:
60:         unplayed_dev_cards = filter(
61:             lambda dc: not dc.played, self.development_cards)
62:
63:         return unplayed_dev_cards
64:
65:     # TODO: pay for placing structure
66:     def get_structure(self, structure_name):
67:
68:         """Get the given structure from the player's stock, if any remains.
69:
70:         Every time a player builds a structure, we need to remove from their
71:         stock, e.g. remaining_road_count etc. This method generalizes this
72:         process of removal for all structures.
73:
74:         Args:
75:             structure_name (str): Class of structure to build.
76:
77:         """
78:
79:         structure_count = self.remaining_structure_counts[structure_name]
80:
81:         if structure_count > 0:
82:             self.remaining_structure_counts[structure_name] -= 1
83:
84:             # TODO: conversions between underscore and camel case
85:             config_path = 'game.structure.player_built.' + structure_name.lower()
86:             structure_dict = Config.get(config_path)
87:
88:             return Structure(self, **structure_dict)
89:         else:
90:             raise NotEnoughStructuresException(self, structure_name)
91:
92:     # TODO: Restore cost of structure
93:     def restore_structure(self, structure_name):
94:         self.remaining_structure_counts[structure_name] += 1
```



```
1: # -*- coding: utf-8 -*-
2: from enum import Enum
3:
4:
5: class PositionType(Enum):
6:
7:     VERTEX = 'vertex'
8:     EDGE = 'edge'
9:
10:     def __str__(self):
11:         return '{0}'.format(self.value)
12:
13:     def __eq__(self, other):
14:         return self.value == other
15:
16:     @classmethod
17:     def find_by_value(cls, value):
18:         """Find the PositionType of the given value."""
19:
20:         for position in cls:
21:             if value == position:
22:                 return position
```



```
1: # -*- coding: utf-8 -*-
2: import random
3: from enum import Enum
4:
5:
6: class ResourceType(Enum):
7:     """Defines the resource types available in a game of Settlers of Catan.
8:
9:     Resources are produced by GameTile's of the given resource type, and are
10:    used to build/buy structures, cards, etc.
11:    """
12:
13:    # Arable tiles are non-fallow tiles.
14:    GRAIN = 'grain'
15:    LUMBER = 'lumber'
16:    WOOL = 'wool'
17:    ORE = 'ore'
18:    BRICK = 'brick'
19:
20:    FALLOW = 'fallow'
21:
22:    def __str__(self):
23:        return '{0}'.format(self.value)
24:
25:    def __eq__(self, other):
26:        return self.value == other
27:
28:    @classmethod
29:    def get_priority_arable_types(cls):
30:
31:        return cls.GRAIN, cls.LUMBER, cls.WOOL, cls.ORE, cls.BRICK
32:
33:    @classmethod
34:    def get_arable_types(cls):
35:        """Get a list of non-fallow ResourceTypes only."""
36:
37:        arable_types = filter(
38:            lambda resource_type: resource_type != ResourceType.FALLOW,
39:            list(ResourceType)
40:        )
41:
42:        return arable_types
43:
44:    @classmethod
45:    def iter_arable_types(cls):
46:        """Returns a generator over non-fallow enum members."""
47:
48:        for resource_type in ResourceType.get_arable_types():
49:            yield resource_type
50:
51:    @classmethod
52:    def random_arable_type(cls):
53:        """Return a random non-fallow ResourceType."""
54:
55:        arable_types = ResourceType.get_arable_types()
56:        random_index = random.randint(0, len(arable_types))
57:
58:        return arable_types[random_index]
59:
60:    @classmethod
61:    def find_by_value(cls, value):
62:        """Find the ResourceType of the given value."""
63:
64:        for resource in cls:
65:            if value == resource:
66:                return resource
```


1:


```
1: # -*- coding: utf-8 -*-
2: from engine.src.config.config import Config
3: from engine.src.lib.utils import Utils
4:
5: class Structure(object):
6:     """
7:     Attributes:
8:         owning_player
9:         name
10:        cost
11:        point_value
12:        extends
13:        upgrades
14:    """
15:
16:    def __init__(self, owning_player, **kwargs):
17:
18:        # Initialize default values.
19:        Config.init_from_config(self, 'game.structure.player_built.default')
20:
21:        # Overwrite default values with custom values.
22:        Utils.init_from_dict(self, kwargs)
23:
24:        self.owning_player = owning_player
25:
26:    def augments(self):
27:        if self.is_augmenting_structure():
28:            return self.upgrades if self.upgrades else self.extends
29:        return None
30:
31:    def is_augmenting_structure(self):
32:        return self.extends or self.upgrades
33:
34:    def __str__(self):
35:        return '{} owned by {}'.format(self.name, self.owning_player)
```



```

1: # -*- coding: utf-8 -*-
2: from engine.src.tile.hex_tile import HexTile
3: from engine.src.resource_type import ResourceType
4: from engine.src.structure.structure import Structure
5:
6:
7: class GameTile(HexTile):
8:     """A hex tile as used in a game of Settlers of Catan.
9:
10:     Args:
11:         resource (ResourceType): The resource/terrain of this hex.
12:
13:         chit_value (int): The value of the chit (i.e. the circular number token)
14:             to be placed on this hex.
15:
16:         calamities (list): A list of calamity objects placed on this tile i.e.
17:             whose passive effects currently affect this tile.
18:     """
19:
20:     def __init__(self, x, y,
21:                 resource_type=ResourceType.FALLOW, chit_value=0):
22:
23:         super(GameTile, self).__init__(x, y)
24:
25:         self.resource_type = resource_type
26:         self.chit_value = chit_value
27:         self.calamities = []
28:
29:     def __str__(self):
30:         return '({0}, {1}) {2} {3}'.format(self.x, self.y,
31:                                           self.resource_type, self.chit_value)
32:
33:     def __repr__(self):
34:         return self.__str__()
35:
36:     def get_adjacent_vertex_structures(self):
37:         """Return any vertices that are structures."""
38:
39:         return filter(
40:             lambda vertex: issubclass(vertex.__class__, Structure),
41:             list(self.iter_vertices())
42:         )
43:
44:     def remove_calamity(self, calamity):
45:         """Remove a calamity from this tile.
46:
47:         Args:
48:             calamity (Calamity): A calamity currently positioned on, and
49:                 affecting, this tile, that will be removed.
50:         """
51:
52:         self.calamities = filter(
53:             lambda existing_calamity: calamity != existing_calamity,
54:             self.calamities
55:         )
56:
57:     def add_calamity(self, calamity):
58:         """Add a calamity to this tile.
59:
60:         Args:
61:             calamity (Calamity): A calamity that, after calling this method,
62:                 will be positioned on, and affect, this tile. The calamity to be
63:                 added.
64:
65:         Returns:
66:             boolean. Whether or not calamity was successfully added. Won't be

```

```

67:         successfully added if had already been placed on this tile.
68:     """
69:
70:     if calamity in self.calamities:
71:         return False
72:     else:
73:         self.calamities.append(calamity)
74:         return True
75:
76:     def get_calamity_tile_placement_effects(self):
77:         """Get a list of tile placement effects for this tile's calamities."""
78:
79:         return filter(
80:             lambda effect: effect is not None,
81:             map(lambda calamity: calamity.tile_placement_effect,
82:                self.calamities)
83:         )

```



```

1: # -*- coding: utf-8 -*-
2:
3: from engine.src.exceptions import *
4: from .tile import Tile
5: from engine.src.vertex import Vertex
6: from engine.src.edge import Edge
7: from engine.src.direction.vertex_direction import VertexDirection
8: from engine.src.direction.edge_vertex_mapping import EdgeVertexMapping
9:
10:
11: class HexTile(Tile):
12:     """A hexagonal tile, with 6 edges and 6 vertices.
13:
14:     Attributes:
15:         vertices (dict): The 6 vertices of this tile, indexed by the
16:             VertexDirection of the vertex i.e. the tuple of the direction,
17:             not its string name.
18:
19:         edges (dict): The edges of this tile, indexed by a pair of vertex
20:             directions.
21:             Note that edges are undirected so edges[src][dst] = edges[dst][src].
22:
23:     Args:
24:         x (int): The x-coordinate of this tile in the axial coordinate system
25:             used by the board to which this tile belongs.
26:
27:         y (int): The y-coordinate of this tile in the axial coordinate system
28:             used by the board to which this tile belongs.
29:
30:     TODO: x and y are mostly here for testing purposes. Removable.
31:     """
32:
33:     def __init__(self, x, y):
34:         self.x = x
35:         self.y = y
36:
37:         self.vertices = {}
38:         self.edges = {}
39:         self._create_vertices_and_edges()
40:
41:     def __repr__(self):
42:         return '({0}, {1})'.format(self.x, self.y)
43:
44:     def __str__(self):
45:         return '({0}, {1})'.format(self.x, self.y)
46:
47:     def _create_vertices_and_edges(self):
48:         """Create brand new vertices and edges for this tile."""
49:
50:         self.vertices = {}
51:         self.edges = {}
52:
53:         for (start_vertex_dir, end_vertex_dir) in VertexDirection.pairs():
54:
55:             end_vertex = Vertex()
56:             self.vertices[end_vertex_dir] = end_vertex
57:
58:             self.add_edge(start_vertex_dir, end_vertex_dir)
59:
60:     def add_edge(self, start_vertex_dir, end_vertex_dir, edge=Edge()):
61:         """Add an edge connecting vertices at given directions to this tile.
62:
63:         Since edges aren't directed, edges[src][dst] = edges[dst][src].
64:
65:     Args:
66:         start_vertex_dir (VertexDirection): Direction relative to

```

```

67:         this tile to the vertex that comprises one end of the edge to add.
68:
69:         end_vertex_dir (VertexDirection): Direction relative to
70:         this tile of the edge-to-add's endpoint vertex.
71:
72:     Returns:
73:         None.
74:
75:     TODO: enforce that these are adjacent vertex directions.
76:     """
77:
78:     if start_vertex_dir not in self.edges:
79:         self.edges[start_vertex_dir] = {}
80:
81:     if end_vertex_dir not in self.edges:
82:         self.edges[end_vertex_dir] = {}
83:
84:     self.edges[start_vertex_dir][end_vertex_dir] = edge
85:     self.edges[end_vertex_dir][start_vertex_dir] = edge
86:
87:     def update_common_edge_and_vertices(self, edge_direction,
88:                                         neighboring_tile):
89:         """Update vertices and edges this tile shares with the neighboring tile.
90:
91:     Args:
92:         edge_direction (EdgeDirection): The given neighboring tile
93:             should share an edge at the given direction relative to this tile.
94:
95:         neighboring_tile (Tile): The tile whose relevant vertices and
96:             edges we should use to overwrite those of this tile.
97:
98:     Returns:
99:         None.
100:     """
101:     # Get the directions of the vertices comprising the endpoints of the
102:     # edge in the given edge_direction i.e. the edge shared between this
103:     # tile and the neighbor tile.
104:     start_vertex_dir, end_vertex_dir = \
105:         EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_direction)
106:
107:     # Get the symmetric directions for the neighbor tile.
108:     neighbor_start_vertex_dir, neighbor_end_vertex_dir = \
109:         EdgeVertexMapping.get_vertex_dirs_for_edge_dir(
110:             edge_direction.get_opposite_direction())
111:
112:     # Get the vertices belonging to the neighboring tile at the found
113:     # directions.
114:     start_vertex = neighboring_tile.vertices[neighbor_start_vertex_dir]
115:     end_vertex = neighboring_tile.vertices[neighbor_end_vertex_dir]
116:
117:     # Replace this tile's vertices with the neighbor's vertices.
118:     self.vertices[start_vertex_dir] = start_vertex
119:     self.vertices[end_vertex_dir] = end_vertex
120:
121:     # Replace this tile's edge with the neighbor's edge.
122:     self.add_edge(start_vertex_dir, end_vertex_dir,
123:                 neighboring_tile.edges[start_vertex_dir][end_vertex_dir])
124:
125:     def iter_edges(self):
126:         """Iterate over the edges of this tile."""
127:
128:         for (start_vertex_dir, end_vertex_dir) in VertexDirection.pairs():
129:             yield self.vertices[start_vertex_dir][end_vertex_dir]
130:
131:     def iter_vertices(self):
132:         """Iterate over the vertices of this tile."""

```

```
133:
134:     for vertex_direction in VertexDirection:
135:         yield self.vertices[vertex_direction]
136:
137: def update_vertex(self, vertex_direction, vertex_value):
138:     """Update the vertex defined by the given vertex direction."""
139:
140:     self.vertices[vertex_direction] = vertex_value
141:
142: @classmethod
143: def get_equivalent_vertex_dir(cls, vertex_dir, edge_dir):
144:     """Get the equivalent vertex as the given one, relative to this tile.
145:
146:     Consider two adjacent tiles, one of which we will think of as the
147:     base_tile, relative to which vertex_dir and edge_dir are defined,
148:     and its neighboring adj_tile. If we know the direction of a vertex
149:     relative to base_tile, and we want to find the direction to the same
150:     vertex relative to adj_tile, we should use this method.
151:
152:     Args:
153:         vertex_dir (VertexDirection): See above.
154:
155:         edge_dir (EdgeDirection): Edge direction of the shared edge,
156:         relative to the given tile, of the edge shared by base_tile and
157:         adj_tile, as described above.
158:
159:     Returns
160:         VertexDirection.
161:     """
162:
163:     # Get the vertex directions, relative to this tile, of the vertices
164:     # that comprise the endpoints of the given edge_dir. Since edge_dir is
165:     # relative to the base_tile, we must find it's opposite to find the
166:     # edge_dir relative to this tile.
167:     opposite_edge_vertices = \
168:         EdgeVertexMapping.get_vertex_dirs_for_edge_dir(
169:             edge_dir.get_opposite_direction())
170:
171:     # Filter out the vertex that is opposite the given vertex, since that
172:     # will not correspond to the same vertex relative to this tile.
173:     vertex = next(vertex for vertex in opposite_edge_vertices if
174:                   vertex != vertex_dir.get_opposite_direction())
175:
176:     return vertex
177:
178: def get_vertex(self, vertex_dir):
179:
180:     if vertex_dir in self.vertices:
181:         return self.vertices[vertex_dir]
182:     else:
183:         raise NoSuchVertexException(self, vertex_dir)
184:
185: def get_edge(self, edge_dir):
186:
187:     vert_src_dir, vert_dst_dir = \
188:         EdgeVertexMapping.get_vertex_dirs_for_edge_dir(edge_dir)
189:
190:     if vert_src_dir in self.edges:
191:         if vert_dst_dir in self.edges[vert_src_dir]:
192:             return self.edges[vert_src_dir][vert_dst_dir]
193:
194:     raise NoSuchEdgeException(self, edge_dir)
```


./engine/src/tile/tile.py

Mon Mar 30 15:15:16 2015

1

```
1: # -*- coding: utf-8 -*-  
2:  
3:  
4: class Tile(object):  
5:     pass
```



```

1: # -*- coding: utf-8 -*-
2: import random
3:
4: from engine.src.config.config import Config
5: from engine.src.trading.trading_entity import TradingEntity
6: from engine.src.trading.trade_offer import TradeOffer
7: from engine.src.exceptions import *
8: from engine.src.card.development_card import DevelopmentCard
9:
10:
11: class Bank(TradingEntity):
12:     """Represents the bank of all available resource cards.
13:
14:     Attributes:
15:         resources (dict): See TradingEntity.
16:
17:         development_cards (list): A list of different development card objects.
18:
19:     Args:
20:         tile_count (int): Number of tiles for the board this bank will be used
21:             with.
22:     """
23:
24:     def __init__(self, tile_count=None):
25:         if tile_count is None:
26:             tile_count = Config.get('game.board.tile_count')
27:
28:         super(Bank, self).__init__()
29:
30:         self.development_cards = []
31:
32:         self._default_init_development_cards()
33:         self._default_init_resources(tile_count)
34:
35:     def _default_init_resources(self, tile_count):
36:         """Determine the initial resources for the bank.
37:
38:         Though not officially a rule, one notices that the default card
39:         allocation for the base game is such that there is, for each resource
40:         type, the same number of cards as there are tiles on the board. In
41:         order to make this function work for different size boards, this is
42:         the rule used to default allocate resource types.
43:
44:     Args:
45:         tile_count (int): Number of tiles on the playing board.
46:
47:     Returns:
48:         None. Modifies self.resources.
49:     """
50:
51:         super(Bank, self)._default_init_resources(tile_count)
52:
53:     def _default_init_development_cards(self):
54:         """Add a configured number of each development card type to the bank."""
55:
56:         dev_card_dict = Config.get('game.card.development')
57:
58:         for name, card in dev_card_dict.iteritems():
59:             for _ in range(card['count']):
60:                 dev_card = DevelopmentCard(**card)
61:                 self.development_cards.append(dev_card)
62:
63:         random.shuffle(self.development_cards)
64:
65:     def buy_development_card(self, player):
66:         """Let the given player purchase a development card from the bank."""

```

```

67:
68:         if not self.development_cards:
69:             raise NotEnoughDevelopmentCardsException
70:
71:         card = self.development_cards.pop()
72:
73:         # Create a trade offer where there are no requested resources,
74:         # just offered resources (cost of development card).
75:         trade_offer = TradeOffer(card.cost, {})
76:
77:         obstructing_entity, obstructing_resource_type = \
78:             trade_offer.validate(player, self)
79:
80:         # If the trade offer is valid, transfer the cost cards and give
81:         # the player the development card.
82:         if not obstructing_entity and not obstructing_resource_type:
83:             trade_offer.execute(player, self)
84:             player.development_cards.append(card)
85:             return card
86:         # Otherwise, return the development card to the deck.
87:         else:
88:             self.development_cards.append(card)
89:             raise NotEnoughResourcesException(obstructing_entity, obstructing_resour
ce_type)

```



```
1: # -*- coding: utf-8 -*-
2: from engine.src.trading.trading_intermediary import TradingIntermediary
3:
4:
5: class Harbor(TradingIntermediary):
6:     """Represents a trading harbor in Settlers of Catan.
7:
8:     Attributes:
9:         supplier (TradingEntity): See TradingIntermediary.
10:
11:         trade_criteria (TradeCriteria): A rule that must be followed for a
12:         trade conducted through this harbor to be considered valid.
13:     """
14:
15:     def __init__(self, supplier, trade_criteria):
16:
17:         super(Harbor, self).__init__(supplier)
18:         self.trade_criteria = trade_criteria
19:
20:     def trade(self, other_entity, trade_offer):
21:         """Attempt to execute the trade only if it follows the trade criteria.
22:
23:         Args:
24:             See TradingIntermediary for:
25:             other_entity (TradingEntity)
26:             trade_offer (TradeOffer)
27:
28:         Returns:
29:             None.
30:         """
31:
32:         if self.trade_criteria.permits(trade_offer):
33:             super(Harbor, self).trade(other_entity, trade_offer)
```



```

1: # -*- coding: utf-8 -*-
2: from enum import Enum
3: from engine.src.resource_type import ResourceType
4:
5:
6: class TradeOffer(object):
7:     # TODO: Convert resources to collections.Counter
8:
9:     def __init__(self, offered_resources, requested_resources):
10:
11:         self.requested_resources = TradeOffer._get_empty_resources()
12:         self.requested_resources.update(requested_resources)
13:
14:         self.offered_resources = TradeOffer._get_empty_resources()
15:         self.offered_resources.update(offered_resources)
16:
17:     @staticmethod
18:     def _get_empty_resources():
19:
20:         resources = {}
21:
22:         for arable_type in ResourceType.get_arable_types():
23:             resources[arable_type] = 0
24:
25:         return resources
26:
27:     def validate(self, proposing_entity, receiving_entity):
28:         """See if this trade can be carried out between the given entities.
29:
30:         Args:
31:             proposing_entity (TradingEntity): The entity that proposed the
32:             trade, i.e. that wants to give the offered_resources and receive
33:             the requested_resources of this trade.
34:
35:             receiving_entity (TradingEntity): The other entity to whom this
36:             trade was proposed and who will receive the offered_resources and
37:             give the requested_resources.
38:
39:         Returns:
40:             TradingEntity, ResourceType. If the trade cannot be completed, this
41:             method returns the entity that is blocking it and the resource
42:             they lack. If the trade can be completed, it will return None.
43:         """
44:
45:         # Check that the proposing_entity has all the resources listed in this
46:         # trade's offered_resources dict.
47:         for resource_type, count in self.offered_resources.iteritems():
48:             if proposing_entity.resources[resource_type] < count:
49:                 return proposing_entity, resource_type
50:
51:         # Check that the receiving entity has all the resources listed in this
52:         # trade's requested_resources dict.
53:         for resource_type, count in self.requested_resources.iteritems():
54:             if receiving_entity.resources[resource_type] < count:
55:                 return receiving_entity, resource_type
56:
57:         return None, None
58:
59:     def execute(self, proposing_entity, receiving_entity):
60:         """Execute this trade based on the given trade entities.
61:
62:         This call should always be preceded by a call to self.validate().
63:
64:         Args:
65:             See self.validate()
66:

```

```

67:         Returns:
68:             None.
69:         """
70:
71:         # Take the offered resources from the entity that proposed the deal
72:         # and give them to the entity that accepted the deal.
73:         for resource_type, count in self.offered_resources.iteritems():
74:             proposing_entity.withdraw_resources(resource_type, count)
75:             receiving_entity.deposit_resources(resource_type, count)
76:
77:         # Take the resources requested by the proposing entity from the
78:         # entity that accepted the deal and give them to the proposing entity.
79:         for resource_type, count in self.requested_resources.iteritems():
80:             proposing_entity.deposit_resources(resource_type, count)
81:             receiving_entity.withdraw_resources(resource_type, count)
82:
83:
84: class TradeMetaCriteria(Enum):
85:     ANY = 1
86:     SAME = 2
87:
88:
89: class TradeCriteria(TradeOffer):
90:     """Defines different trade criteria."""
91:
92:     def __init__(self, offered_resources=None, requested_resources=None,
93:                  offered_meta=None, requested_meta=None):
94:
95:         super(TradeCriteria, self).__init__(offered_resources,
96:                                              requested_resources)
97:
98:         self.offered_meta = TradeCriteria._get_empty_meta()
99:         self.requested_meta = TradeCriteria._get_empty_meta()
100:
101:         self.offered_meta.update(offered_meta)
102:         self.requested_meta.update(requested_meta)
103:
104:     @staticmethod
105:     def _get_empty_meta():
106:
107:         meta = {}
108:
109:         for criteria in TradeMetaCriteria:
110:             meta[criteria] = 0
111:
112:         return meta
113:
114:     def permits(self, trade_offer):
115:
116:         valid_offer = self.valid(self.offered_resources, self.offered_meta,
117:                                  trade_offer.offered_resource)
118:
119:         valid_req = self.valid(self.requested_resources, self.requested_meta,
120:                                trade_offer.requested_resources)
121:
122:         return valid_offer and valid_req
123:
124:     @staticmethod
125:     def valid(crit_resources, crit_meta, offered_resources):
126:
127:         offered_resources = offered_resources.copy()
128:
129:         valid = True
130:
131:         # First handle meta
132:         if valid and TradeMetaCriteria.SAME in crit_meta:

```

```
133:
134:         valid = False
135:
136:         req_same_resource_count = crit_meta[TradeMetaCriteria.SAME]
137:
138:         for resource_type, count in offered_resources.iteritems():
139:             if count >= req_same_resource_count:
140:                 offered_resources[resource_type] -= req_same_resource_count
141:                 valid = True
142:                 break
143:
144:         if valid and TradeMetaCriteria.ANY in crit_meta:
145:
146:             req_any_resource_count = crit_meta[TradeMetaCriteria.ANY]
147:
148:             for resource_type, count in offered_resources.iteritems():
149:                 if count > 0:
150:                     deduct = min(count, req_any_resource_count)
151:
152:                     req_any_resource_count -= deduct
153:                     offered_resources[resource_type] -= deduct
154:
155:             if req_any_resource_count > 0:
156:                 valid = False
157:
158:         if valid:
159:             # Now handle normal resources
160:             for resource_type, count in crit_resources.iteritems():
161:                 if count != offered_resources[resource_type]:
162:                     valid = False
163:
164:         return valid
```



```

1: # -*- coding: utf-8 -*-
2: import random
3: from collections import Counter
4: from engine.src.lib.utils import Utils
5: from engine.src.exceptions import NotEnoughResourcesException
6: from engine.src.resource_type import ResourceType
7: from engine.src.trading.trade_offer import TradeOffer
8:
9:
10: class TradingEntity(object):
11:     """Represents an entity capable of storing and trading resources.
12:
13:     Attributes:
14:         resources (dict): Represents all resources currently owned by this
15:             entity. Keys are arable ResourceTypes and values are integers
16:             representing the amount of a particular resource type the entity has.
17:
18:     TODO: This should be an abstract class.
19:     """
20:
21:     def __init__(self):
22:         self.resources = {}
23:         # TODO: Freak error where Python isn't recognizing default arg.
24:         self._default_init_resources(0)
25:
26:     def _default_init_resources(self, count):
27:         """Initialize this entity to have count resources per resource type.
28:
29:     Args:
30:         count (int): Number of each arable resource this entity will have.
31:
32:     Returns:
33:         None. Modifies self.resources.
34:     """
35:
36:         self.resources = {}
37:         for arable_type in ResourceType.get_arable_types():
38:             self.resources[arable_type] = count
39:
40:     def count_resources(self):
41:         return sum(self.resources.values())
42:
43:     def validate_resources(self, resources):
44:         """Check that this player has at least as many resources as given."""
45:
46:         default_resources = TradeOffer._get_empty_resources()
47:         default_resources.update(resources)
48:
49:         resources = default_resources
50:
51:         # This entity does not have the given resources if the difference
52:         # between its count and the given resources dict count for any given
53:         # resource type is negative.
54:         resource_debt = {resource_type: count - resources[resource_type]
55:                         for resource_type, count in self.resources.items()
56:                         if count - resources[resource_type] < 0}
57:
58:         valid = len(resource_debt.keys()) == 0
59:
60:         if valid:
61:             return True
62:         else:
63:             raise NotEnoughResourcesException(self, resource_debt.keys())
64:
65:     def get_resource_list(self):
66:         """Get a list of resource types, one for each "card" this player has."""

```

```

67:
68:     return Utils.flatten(map(
69:         lambda resource_type:
70:             [resource_type] * self.resources[resource_type],
71:         self.resources
72:     ))
73:
74:
75:     def transfer_resources(self, to_entity, resource_type, resource_count):
76:         """Transfer specified resources from this entity to the given entity."""
77:
78:         self.withdraw_resources(resource_type, resource_count)
79:         to_entity.deposit_resources(resource_type, resource_count)
80:
81:     def withdraw_resources(self, resource_type, resource_count):
82:         """Withdraw the specified number of resources from the entity.
83:
84:     Args:
85:         resource_type (ResourceType): Type of resource to withdraw.
86:
87:         resource_count (int): Number of resources of the given type to
88:             withdraw.
89:
90:     Raises:
91:         NotEnoughResourcesException. When the withdrawal is for more
92:             resources than the entity currently has.
93:     """
94:
95:     if resource_type == ResourceType.FALLOW:
96:         # TODO: raise exception.
97:         return
98:
99:     if self.resources[resource_type] >= resource_count:
100:         self.resources[resource_type] -= resource_count
101:     else:
102:         raise NotEnoughResourcesException(self, resource_type)
103:
104:     def withdraw_random_resource(self):
105:         """Remove a random resource from this trading entity.
106:
107:     Note that this method only withdraws a single random resource.
108:     Callers of this method should check to make sure that this entity
109:     still has resources using self.count_resources().
110:     """
111:
112:     resources = self.get_resource_list()
113:
114:     resource_type = random.choice(resources)
115:
116:     self.resources[resource_type] -= 1
117:
118:     return resource_type
119:
120:     def deposit_multiple_resources(self, resource_type_count_dict):
121:         for resource_type, count in resource_type_count_dict.iteritems():
122:             self.deposit_resources(resource_type, count)
123:
124:     def deposit_resources(self, resource_type, resource_count):
125:         """Deposit the specified number of resources from the entity.
126:
127:     Args:
128:         resource_type (ResourceType): Type of resource to deposit.
129:
130:         resource_count (int): Number of resources of the given type to
131:             deposit.
132:     """

```

```
133:
134:         if resource_type != ResourceType.FALLOW:
135:             self.resources[resource_type] += resource_count
136:
137:     def trade(self, requesting_entity, trade_offer):
138:         """Trade one resource for another at a given ratio.
139:
140:         Args:
141:             requesting_entity (TradingEntity): Entity who has proposed a trade
142:             wherein they offer the trade's offered_resources and request the
143:             trade's requested_resources from this entity.
144:
145:             trade (Trade): Keeps track of how many of which resource are being
146:             offered and requested.
147:
148:         Raises:
149:             NotEnoughResourcesException. When this or the other entity lacks
150:             the resources to complete the trade.
151:         """
152:
153:         obstructing_entity, obstructing_resource_type = \
154:             trade_offer.validate(requesting_entity, self)
155:
156:         if obstructing_entity is not None:
157:             raise NotEnoughResourcesException(obstructing_entity,
158:                                                obstructing_resource_type)
159:
160:         else:
161:             trade_offer.execute(requesting_entity, self)
```

```
1: # -*- coding: utf-8 -*-
2: from engine.src.trading.trading_entity import TradingEntity
3:
4:
5: class TradingIntermediary(object):
6:     """Represents an entity capable of trading resources on behalf of two other
7:     TradingEntity's, but incapable of storing resources itself.
8:
9:     Args:
10:         supplier (TradingEntity): The entity who owns the resources this
11:         intermediary is allowed to trade on its behalf.
12:     """
13:
14:     def __init__(self, supplier):
15:
16:         if not isinstance(supplier, TradingEntity):
17:             message = 'Invalid trading entity given as supplier'
18:             raise ValueError(message)
19:
20:         self.supplier = supplier
21:
22:     def trade(self, other_entity, trade_offer):
23:         """Attempt to execute the given trade.
24:
25:         Args:
26:             other_entity (TradingEntity): Entity that proposed the trade to
27:             the harbor.
28:
29:             trade_offer (TradeOffer): Trade offer crafted by the other entity.
30:
31:         Returns:
32:             None.
33:         """
34:
35:         self.supplier.trade(other_entity, trade_offer)
```



```
1: class AsciiHexBoard(object):
2:
3:     board_string = \
4:     """
5:         / \ / \ / \ / \
6:         /   \   \   \   \
7:         | -2,2 | -1, 2 | 0,2 |
8:         |     |     |     |
9:         / \ / \ / \ / \
10:        /   \   \   \   \
11:        | -2,1 | -1,1 | 0, 1 | 1,1 |
12:        |     |     |     |
13:        / \ / \ / \ / \
14:       /   \   \   \   \
15:       | -2,0 | -1,0 | 0,0 | 1,0 | 2,0 |
16:       |     |     |     |
17:       / \ / \ / \ / \
18:      /   \   \   \   \
19:      | -1,-1 | 0,-1 | 1,-1 | 2,-1 |
20:      |     |     |     |
21:      / \ / \ / \ / \
22:     /   \   \   \   \
23:     | 0,-2 | 1,-2 | 2,-2 |
24:     |     |     |     |
25:     / \ / \ / \ / \
26:    /   \   \   \   \
27:    """
```



```
1: # -*- coding: utf-8 -*-
2: from abc import ABCMeta
3:
4:
5: class Vertex(object):
6:     __metaclass__ = ABCMeta
```



```
1: # TODO: Cleanup. Separate module registration with game run logic?
2:
3: # Add engine package to Python path.
4: import sys
5: import os
6:
7: sys.path.insert(1, os.path.dirname(os.path.dirname(os.path.abspath(__file__))))
8:
9:
10: # Catch SIGINT for prettier force quit handling.
11: import signal
12:
13: def signal_handler(signal, frame):
14:     print '\nYou force quit the game.'
15:     sys.exit(0)
16:
17: signal.signal(signal.SIGINT, signal_handler)
18:
19:
20: # Run main game loop.
21: from engine.src.game import Game
22: from engine.src.config.config import Config
23:
24: print Config.get('game.board.tile_count')
25:
26: # g = Game()
27: # g.start()
28:
```



```
1: """
2: A pretty-printing dump function for the ast module. The code was copied from
3: the ast.dump function and modified slightly to pretty-print.
4:
5: Alex Leone (acleone ~AT~ gmail.com), 2010-01-30
6: """
7:
8: from ast import *
9:
10: def dump(node, annotate_fields=True, include_attributes=False, indent=' '):
11:     """
12:     Return a formatted dump of the tree in *node*. This is mainly useful for
13:     debugging purposes. The returned string will show the names and the values
14:     for fields. This makes the code impossible to evaluate, so if evaluation is
15:     wanted *annotate_fields* must be set to False. Attributes such as line
16:     numbers and column offsets are not dumped by default. If this is wanted,
17:     *include_attributes* can be set to True.
18:     """
19:     def _format(node, level=0):
20:         if isinstance(node, AST):
21:             fields = [(a, _format(b, level)) for a, b in iter_fields(node)]
22:             if include_attributes and node._attributes:
23:                 fields.extend([(a, _format(getattr(node, a), level))
24:                                for a in node._attributes])
25:             return ''.join([
26:                 node.__class__.__name__,
27:                 '(',
28:                 ', '.join('%s=%s' % field for field in fields)
29:                 if annotate_fields else
30:                 (b for a, b in fields)),
31:                 ')'])
32:         elif isinstance(node, list):
33:             lines = ['[']
34:             lines.extend((indent * (level + 2) + _format(x, level + 2) + ', '
35:                                                             for x in node))
36:             if len(lines) > 1:
37:                 lines.append(indent * (level + 1) + ']')
38:             else:
39:                 lines[-1] += ']'
40:             return '\n'.join(lines)
41:         return repr(node)
42:     if not isinstance(node, AST):
43:         raise TypeError('expected AST, got %r' % node.__class__.__name__)
44:     return _format(node)
45:
46: if __name__ == '__main__':
47:     import sys
48:     for filename in sys.argv[1:]:
49:         print '=' * 50
50:         print 'AST tree for', filename
51:         print '=' * 50
52:         f = open(filename, 'r')
53:         fstr = f.read()
54:         f.close()
55:         print dump(parse(fstr, filename=filename), include_attributes=True)
56:         print
```



```

1: from collections import defaultdict
2:
3: def get_registry():
4:     """Produces a registration decorator that allows methods to be gathered under ta
gs
5:     """
6:     registry = defaultdict(list)
7:     def register(nonterminal):
8:         def registrar(func):
9:             registry[nonterminal] += [func]
10:            return func
11:        return registrar
12:    register.get = lambda x: registry[x]
13:    return register
14:
15: def gen_grammar(name, nonterminals, indent=4):
16:     """Generates a grammar docstring for the provided name and nonterminals
17:     E.x. name : nonterminal1
18:         | nonterminal2
19:
20:     Args:
21:         name (String): The nonterminal name
22:         nonterminals (List): A list of the nonterminals it's associated with
23:
24:     Returns:
25:         String. A docstring representing the grammar of the nonterminal
26:     """
27:     docstring = "{ } : {}".format(name, nonterminals[0])
28:     padding = ' ' * (len(name) + 1 + indent) + '| '
29:
30:     if len(nonterminals) > 1:
31:         docstring += '\n' + padding + ('\n' + padding).join(nonterminals[1:])
32:
33:     return docstring
34:
35: def trivial(name, nonterminals, indent=4, suffix=''):
36:     """Generates a method for a trivial terminal, where p[0] = p[1]
37:
38:     Args:
39:         name (String): A string representing the nonterminal name
40:         nonterminals (List): A list of strings representing the nonterminals it's li
nked to
41:
42:     Named Args:
43:         indent (Int): 4 -- An int representing the amount of indentation in the file
44:         suffix (String): '' -- A string representing a suffix that should be added t
o the name of the function
45:
46:     Returns:
47:         Func. A function with the provided name and a generated grammar docstring
48:     """
49:     def template(p):
50:         p[0] = p[1]
51:
52:     template.__doc__ = gen_grammar(name, nonterminals, indent)
53:
54:     template.__name__ = template.func_name = 'p_' + name + suffix
55:
56:     return template
57:
58: def trivial_from_registry(name, registry, indent=4, suffix=''):
59:     """Generates a method for a trivial terminal, where p[0] = p[1], sourcing nonter
minals from a registry
60:
61:     Args:
62:         name (String): A string representing the nonterminal name

```

```

63:         registry (Dict): A registry generated by the get_registry() function
64:
65:     Named Args:
66:         indent (Int): 4 -- An int representing the amount of indentation in the file
67:         suffix (String): '' -- A string representing a suffix that should be added t
o the name of the function
68:
69:     Returns:
70:         Func. A function with the provided name and a generated grammar docstring
71:     """
72:     return trivial(name, [func.__doc__.split(':')[0].strip() for func in registry.ge
t(name)], indent=indent, suffix=suffix)

```



```

1: #import sys
2: #sys.path.append('.')
3: #from ..engine.src.lib.utils import Utils
4: from collections import defaultdict
5:
6: class StateNotFound(Exception):
7:     """Thrown when a dependency injection tries to inject a variable that isn't part
of the declared game state
8:     """
9:     pass
10:
11:
12: class GameOracle(object):
13:     """A wrapper object for the game state, providing a simple interface to isolate
development of the imperative
14:     parser from the game engine
15:     """
16:
17:     def __init__(self, state={}):
18:         """Creates an instance of a GameOracle
19:
20:         Named Args:
21:             state (Dict): {} -- a dictionary containing references from variable nam
e strings to game state objects
22:
23:         Returns:
24:             GameOracle. An oracle which can access the provided state dictionary
25:         """
26:         self.game_state = state
27:
28:     def get(self, name):
29:         """Get a variable from the GameOracle's state
30:
31:         Args:
32:             name (String): A string representing the name of the variable to retriev
e
33:
34:         Returns:
35:             Any. The value of the variable being retrieved
36:
37:         Throws:
38:             StateNotFound -- when a state being accessed isn't present in the state
dict
39:         """
40:         try:
41:             return self.game_state[name]
42:         except KeyError:
43:             raise StateNotFound("Variable \"%s\" not present in game state" % name)
44:
45:     def set(self, name, var):
46:         """Set a particular variable in the state dict to a particular value
47:
48:         Args:
49:             name (String): A string representing the name to store the variable unde
r
50:             var (Any): The value to store for the variable
51:         """
52:         self.game_state[name] = var
53:
54: # Access game state through the game oracle
55: ORACLE = GameOracle(defaultdict(list))

```



```

1: import ast
2: from collections import defaultdict
3:
4: import ply.lex as lex
5: import ply.yacc as yacc
6:
7: from grammar_utils import get_registry, trivial_from_registry, trivial, gen_grammar
8: from utils import flatten, find_column
9:
10: # Allow dependency injection using the predefined GameOracle
11: from oracle import ORACLE
12:
13: class RewriteInjected(ast.NodeTransformer):
14:     def __init__(self, injected):
15:         """Creates a NodeTransformer object to replace calls to injected parameters
with calls to a lookup table
16:
17:     Args:
18:         injected (Iterable): An iterable representing the list of injected parameter names
19:
20:     Returns:
21:         An instance of RewriteInjected whose visit method will rewrite the injected nodes
22:
23:     """
24:         super(RewriteInjected, self).__init__()
25:         self.injected = set(injected)
26:
27:     def visit_Name(self, node):
28:         if node.id in self.injected:
29:             return ast.copy_location(ast.Call(
30:                 ast.Attribute(
31:                     ast.Name('ORACLE', ast.Load()),
32:                     'get', ast.Load()
33:                 ), [ast.Str(node.id)], [], None, None), node)
34:         else:
35:             return self.generic_visit(node)
36:
37: # Automatically build no-op nonterminals
38: register = get_registry()
39:
40: def gen_function(name):
41:     """Generates a function for the given trivial nonterminal based on the registry
42:
43:     Args:
44:         name (String): A string representing the nonterminal to generate the function for
45:
46:     Returns:
47:         Func. A trivial function p[0] = p[1] for the nonterminal
48:
49:     """
50:     return trivial_from_registry(name, register, suffix='_reg')
51:
52: # Helper functions
53:
54: def listify(p, item_pos=1, list_pos=3, size_check=2):
55:     """Creates a list of values from the given nonterminal parse p
56:
57:     Args:
58:         p (List): A list representing the parse
59:
60:     Named Args:
61:         item_pos (Int): 1 -- An int representing the position of the item at the head of the list
62:         list_pos (Int): 3 -- An int representing the position of the rest of the list
63: """

```

```

61:         size_check (Int): 2 -- An int representing the length of the parse of a single item of the list
62:
63:     Returns:
64:         List. The parse p, with p[0] set to the list of items
65:
66:     """
67:     p[0] = [p[item_pos]] if p[item_pos] else []
68:     if len(p) > size_check:
69:         p[0].extend(p[list_pos])
70:     return p
71:
72: # Token declarations
73:
74: # TODO allow reserved words in strings
75: reserved = {k: k.upper() for k in [
76:     'func',
77:     'return',
78:     'print',
79:     'if',
80:     'else',
81:     'or',
82:     'and',
83:     'not',
84:     'while',
85:     'for',
86:     'to'
87: ]}
88:
89: tokens = ['ID', 'NUM', 'COMPOP', 'AUGASSIGN', 'NEWLINE', 'IN', 'STRING'] + list(reserved.values())
90:
91: literals = ['=', '+', '-', '*', '/', '(', ')', '{', '}', '[', ',', ']', '.', '@']
92:
93: def t_STRING(t):
94:     r'\"(\\.|[^\"])*\"|\'(\\.|[^\'])*\''
95:     t.value = t.value.strip('"').strip("'")
96:     return t
97:
98: def t_ID(t):
99:     r'[a-zA-Z_][a-zA-Z0-9_]*'
100:     t.type = reserved.get(t.value, 'ID') # Check for reserved words
101:     return t
102:
103: def t_NUM(t):
104:     r'\d+|\d+\.\d+'
105:     try:
106:         t.value = int(t.value)
107:     except ValueError:
108:         print 'Integer value too large', t.value
109:         t.value = 0
110:     return t
111:
112: t_COMPOP = r'<=>|<|>|!=|
113: t_AUGASSIGN = r'\+=|-=|*=|/=|
114: t_IN = r':='
115:
116: t_ignore = " \t"
117:
118: def t_NEWLINE(t):
119:     r'\n\s+'
120:     t.lexer.lineno += t.value.count('\n')
121:     return t
122:
123: def t_error(t):
124:     print 'Illegal character "%s"' % t.value[0]
125:
126: # Build the lexer
127: lexer = lex.lex()

```

```

125:
126: # Parsing rules
127: precedence = (
128:     ('left', '+', '-'),
129:     ('left', '*', '/'),
130:     ('left', 'OR'),
131:     ('left', 'AND'),
132:     ('left', 'COMPOP'),
133:     ('left', 'TO'),
134:     ('right', 'NOT'),
135:     ('right', 'UMINUS'),
136:     ('right', '('),
137:     ('left', '['),
138:     ('left', '.')
139: )
140:
141: # Simple expressions
142:
143: @register('expr')
144: def p_id(p):
145:     """id : ID"""
146:     p[0] = ast.Name(p[1], ast.Load())
147:
148: def p_store_id(p):
149:     """store_id : ID"""
150:     p[0] = ast.Name(p[1], ast.Store())
151:
152: def p_assign_id(p):
153:     """assign_id : assign_lst"""
154:     p[0] = ast.Tuple(p[1], ast.Store()) if len(p[1]) > 1 else p[1][0]
155:
156: def p_assign_lst(p):
157:     """assign_lst : store_id ',' assign_lst
158:     | store_id"""
159:     p = listify(p)
160:
161: def p_store_property(p):
162:     """store_id : property"""
163:     p[1].ctx = ast.Store()
164:     p[0] = p[1]
165:
166: def p_store_getitem(p):
167:     """store_id : getitem"""
168:     p[1].ctx = ast.Store()
169:     p[0] = p[1]
170:
171: @register('expr')
172: def p_num(p):
173:     """num : NUM"""
174:     p[0] = ast.Num(p[1])
175:
176: # Groupings
177:
178: def p_expr_group(p):
179:     """expr : '(' expr ')'"""
180:     p[0] = p[2]
181:
182: # Strings
183:
184: @register('expr')
185: def p_str(p):
186:     """str : STRING"""
187:     p[0] = ast.Str(p[1])
188:
189: # Statements
190:

```

```

191: def p_stmt_expr(p):
192:     """stmt : expr"""
193:     p[0] = ast.Expr(p[1])
194:
195: def p_stmt_assignment(p):
196:     """stmt : assign_id '=' expr"""
197:     p[0] = ast.Assign([p[1]], p[3])
198:
199: def p_stmt_aug_assignment(p):
200:     """stmt : store_id AUGASSIGN expr"""
201:     symbol_conversions = {
202:         '+=': ast.Add,
203:         '-=': ast.Sub,
204:         '*=': ast.Mult,
205:         '/=': ast.Div
206:     }
207:     p[0] = ast.AugAssign(p[1], symbol_conversions[p[2]](), p[3])
208:
209: def p_stmt_return(p):
210:     """stmt : RETURN expr
211:     | RETURN"""
212:     if len(p) > 2:
213:         p[0] = ast.Return(p[2])
214:     else:
215:         p[0] = ast.Return(None)
216:
217: def p_stmt_print(p):
218:     """stmt : PRINT expr"""
219:     p[0] = ast.Print(None, p[2] if isinstance(p[2], list) else [p[2]], True)
220:
221: # Functions
222:
223: @register('stmt')
224: def p_top_func(p):
225:     """topfunc : FUNC '(' params ')' '{' opt_newline body '}'"""
226:     if p[3]:
227:         args = ast.arguments([ast.Name('self', ast.Param())], None, None, [])
228:     else:
229:         args = ast.arguments([], None, None, [])
230:     p[7] = [RewriteInjected([param[0].id for param in p[3]]).visit(node) for node in
231: p[7]]
232:     p[0] = [ast.FunctionDef("top", args, p[7], [])]
233:
234: @register('stmt')
235: def p_func(p):
236:     """func : FUNC ID '(' params ')' '{' opt_newline body '}'"""
237:     arg_names, defaults = tuple([filter(lambda x: x is not None, item) for item
in zip(*p[4])])
238:     args = ast.arguments(list(arg_names), None, None, list(defaults))
239:     else:
240:         args = ast.arguments([], None, None, [])
241:     p[0] = ast.FunctionDef(p[2], args, p[8], [])
242:
243: @register('expr')
244: def p_funccall(p):
245:     """funcall : expr '(' opt_newline expr_list ')'"""
246:     keywords = filter(lambda x: isinstance(x, ast.keyword), p[4])
247:     exprs = filter(lambda x: not isinstance(x, ast.keyword), p[4])
248:     p[0] = ast.Call(p[1], exprs, keywords, None, None)
249:
250: @register('expr')
251: def p_lambda(p):
252:     """lambda : '@' '(' params ')' expr"""
253:     if p[3]:
254:         arg_names, defaults = tuple([filter(lambda x: x is not None, item) for item

```

```

in zip(*p[3]))
255:     args = ast.arguments(list(arg_names), None, None, list(defaults))
256:     else:
257:         args = ast.arguments([], None, None, [])
258:         p[0] = ast.Lambda(args, p[5])
259:
260: def p_body(p):
261:     """body : stmtlst
262:         | empty"""
263:     if p[1]:
264:         p[0] = p[1]
265:     else:
266:         p[0] = [ast.Pass()]
267:
268: p_opt_newline = trivial('opt_newline', ['NEWLINE', 'empty'])
269:
270: # Boolean logic
271:
272: @register('expr')
273: def p_compare(p):
274:     """compare : expr COMPOP expr"""
275:     symbol_conversions = {
276:         '==': ast.Eq,
277:         '!=': ast.NotEq,
278:         '<=': ast.LtE,
279:         '>=': ast.GtE,
280:         '<': ast.Lt,
281:         '>': ast.Gt
282:     }
283:
284:     p[0] = ast.Compare(p[1], [symbol_conversions[p[2]]()], [p[3]])
285:
286: def p_bool_expr(p):
287:     """expr : expr AND expr
288:         | expr OR expr"""
289:     symbol_conversion = {
290:         'and': ast.And,
291:         'or': ast.Or
292:     }
293:     if isinstance(p[1], ast.BoolOp) and isinstance(p[1].op, symbol_conversion[p[2]]):
294:         p[1].values.append(p[3])
295:         p[0] = p[1]
296:     else:
297:         p[0] = ast.BoolOp(symbol_conversion[p[2]](), [p[1], p[3]])
298:
299: def p_expr_not(p):
300:     """expr : NOT expr %prec NOT"""
301:     p[0] = ast.UnaryOp(ast.Not(), p[2])
302:
303: # Conditionals
304:
305: @register('stmt')
306: def p_if(p):
307:     """if : IF expr '{' opt_newline body '}' opt_else"""
308:     p[0] = ast.If(p[2], p[5], p[7])
309:
310: def p_opt_else(p):
311:     """opt_else : ELSE expr '{' opt_newline body '}'
312:         | empty"""
313:     if len(p) > 2:
314:         p[0] = p[4]
315:     else:
316:         p[0] = []
317:
318: def p_opt_elseif(p):

```

```

319:     """opt_else : ELSE expr '{' opt_newline body '}' opt_else"""
320:     p[0] = [ast.If(p[2], p[5], p[7])]
321:
322: # Loops
323: @register('stmt')
324: def p_while(p):
325:     """while : WHILE expr '{' opt_newline body '}'"""
326:     p[0] = ast.While(p[2], p[5], [])
327:
328: @register('stmt')
329: def p_for(p):
330:     """for : FOR ID IN expr '{' opt_newline body '}'"""
331:     p[0] = ast.For(ast.Name(p[2], ast.Store()), p[4], p[7], [])
332:
333: @register('expr')
334: def p_range(p):
335:     """to : expr TO expr"""
336:     p[0] = ast.Call(ast.Name('range', ast.Load()), [p[1], p[3]], [], None, None)
337:
338: # Lists
339:
340: def p_params(p):
341:     """params : param ',' opt_newline params
342:         | param"""
343:     p = listify(p, list_pos=4)
344:
345: def p_param(p):
346:     """param : ID
347:         | ID '=' expr
348:         | empty"""
349:     if p[1]:
350:         p[0] = (ast.Name(p[1], ast.Param()), None if len(p) < 3 else p[3])
351:
352: def p_stmtlst(p):
353:     """stmtlst : stmt NEWLINE stmtlst
354:         | stmt opt_newline"""
355:     p = listify(p, size_check=3)
356:
357: def p_in_params(p):
358:     """expr_list : opt_expr ',' opt_newline expr_list
359:         | opt_expr"""
360:     p = listify(p, list_pos=4)
361:
362: p_opt_expr = trivial('opt_expr', ['expr', 'empty'])
363:
364: def p_opt_expr_default(p):
365:     """opt_expr : ID '=' expr"""
366:     p[0] = ast.keyword(p[1], p[3])
367:
368: @register('expr')
369: def p_list_braces(p):
370:     """list : '[' expr_list ']'"""
371:     p[0] = ast.List(p[2], ast.Load())
372:
373: # Property access
374:
375: @register('expr')
376: def p_expr_property(p):
377:     """property : expr '.' ID"""
378:
379:     p[0] = ast.Attribute(p[1], p[3], ast.Load())
380:
381: @register('expr')
382: def p_expr_getitem(p):
383:     """getitem : expr '[' expr ']'"""
384:     p[0] = ast.Subscript(p[1], ast.Index(p[3]), ast.Load())

```

```

385:
386: # Arithmetic
387:
388: def p_expr_binop(p):
389:     """expr : expr '+' expr
390:             | expr '-' expr
391:             | expr '*' expr
392:             | expr '/' expr"""
393:     if p[2] == '+': p[0] = ast.BinOp(p[1], ast.Add(), p[3]) # p[1] + p[3]
394:     elif p[2] == '-': p[0] = ast.BinOp(p[1], ast.Sub(), p[3]) # p[1] - p[3]
395:     elif p[2] == '*': p[0] = ast.BinOp(p[1], ast.Mult(), p[3]) # p[1] * p[3]
396:     elif p[2] == '/': p[0] = ast.BinOp(p[1], ast.Div(), p[3]) # p[1] / p[3]
397:
398: def p_expr_uminus(p):
399:     """expr : '-' expr %prec UMINUS"""
400:     if isinstance(p[2], ast.Num):
401:         p[2].n *= -1
402:         p[0] = p[2]
403:     else:
404:         p[0] = ast.UnaryOp(ast.USub(), p[2])
405:
406: # Terminal registration
407:
408: p_expr_reg = gen_function('expr')
409: p_stmt_reg = gen_function('stmt')
410:
411: # Meta terminals
412:
413: # Globals for communicating with p_error
414: # This is a code smell, but I don't think there's any easy way of
415: # communicating this otherwise
416: LINE_OFFSET = 1
417: COL_OFFSET = 1
418: FUNC_STR = ''
419:
420: def p_error(p):
421:     print ' [%d:%d] Syntax error at "%s"' % (p.lineno + LINE_OFFSET - 1, find_column(
422: FUNC_STR, p) + COL_OFFSET - 2, p.value)
423:
424: def p_empty(p):
425:     """empty : """
426:     pass
427:
428: test_parser = yacc.yacc(start='stmtlst')
429: parser = yacc.yacc(start='topfunc')
430:
431: class BadParseException(Exception):
432:     def __init__(self, *args, **kwargs):
433:         super(self, BadParseException).__init__(*args, **kwargs)
434:
435: def parse_string(s, debug=False, testing=False):
436:     """Parses a given string into a Python AST
437:
438:     Args:
439:         s (String): The string to parse into an AST
440:
441:     Named Args:
442:         debug (Bool): False -- A boolean representing whether to print debug info
443:         testing (Bool): False -- A boolean representing whether to use 'stmtlst' or
'topfunc' as the starting symbol
444:
445:     Returns:
446:         ast.Module. The AST representation of the provided code string
447:
448:     """
449:     if testing:
450:         body = test_parser.parse(s.strip(), debug=debug, lexer=lexer)

```

```

449:     else:
450:         body = parser.parse(s.strip(), debug=debug, lexer=lexer)
451:     return ast.Module(body)
452:
453: def parse_function(func_str, name='top', debug=False, line_offset=1, col_offset=1):
454:     """Parses a string representing a Skit function into a first-class Python functi
on
455:
456:     Args:
457:         func_str (String): The string representing a Skit function to parse into a P
ython function
458:
459:     Named Args:
460:         name (String): 'top' -- A string representing the name to give the function
being parsed
461:         debug (Bool): False -- A boolean representing whether to print debug info
462:         line_offset (Int): 0 -- An int representing the line offset at which the fun
ction was found
463:         col_offset (Int): 0 -- An int representing the column offset at which the fu
nction was found
464:
465:     Returns:
466:         Func. A first-class Python function that performs the actions of the Skit fu
nction provided
467:
468:     """
469:     global LINE_OFFSET
470:     global COL_OFFSET
471:     global FUNC_STR
472:     LINE_OFFSET = line_offset
473:     COL_OFFSET = col_offset
474:     FUNC_STR = func_str
475:
476:     func_ast = ast.fix_missing_locations(parse_string(func_str, debug=debug))
477:
478:     exec(compile(func_ast, filename='<ast>', mode='exec'))
479:     locals()[name].__name__ = locals()[name].func_name = name
480:     return locals()[name]
481:
482: env = locals()
483:
484: def print_grammar():
485:     """Prints the grammar formed by the functions in this file
486:
487:     """
488:     p_funcs = [func for name, func in env.items() if
489:                 name.startswith('p_') and
490:                 hasattr(func, '__call__') and
491:                 name != 'p_error']
492:
493:     grammar = defaultdict(list)
494:     for name, nonterminals in [func.__doc__.split(':') for func in p_funcs]:
495:         grammar[name.strip()].append(nonterminals)
496:
497:     grammar = {key: [item for item in flatten(
498:         [[docstr.strip() for docstr in item.split('|')] for item in value]
499:     )] for key, value in grammar.iteritems()}
500:
501:     for name, nonterminals in grammar.iteritems():
502:         print gen_grammar(name, sorted(nonterminals), indent=0) + '\n'
503:
504: if __name__ == '__main__':
505:     while 1:
506:         try:
507:             s = raw_input('>')
508:         except EOFError:
509:             break
510:         if not s: continue
511:         print ast.dump(parse_string(s))

```

```

1: import unittest
2: import ast
3:
4: #TODO fix relative import
5: from ..parser import parse_string, parse_function
6: from ..oracle import ORACLE
7:
8: class ParsingASTTests(unittest.TestCase):
9:     def assertSameParse(self, python, skit):
10:         self.assertEqual(
11:             ast.dump(ast.parse(python)),
12:             ast.dump(parse_string(skit, testing=True))
13:         )
14:
15:     def test_id(self):
16:         self.assertSameParse("test", "test")
17:
18:     def test_num(self):
19:         self.assertSameParse("1", "1")
20:
21:     def test_group(self):
22:         self.assertSameParse("(1 + 2)", "(1 + 2)")
23:
24:     def test_string_single_quotes(self):
25:         self.assertSameParse("'test'", "'test'")
26:
27:     def test_string_double_quotes(self):
28:         self.assertSameParse('"test"', '"test"')
29:
30:     def test_stmt_assignment(self):
31:         self.assertSameParse("test = 1", "test = 1")
32:
33:     def test_multi_stmt_assignment(self):
34:         self.assertSameParse("a, b = tpl", "a, b = tpl")
35:
36:     def test_stmt_assign_property(self):
37:         self.assertSameParse("a.b.c = 1", "a.b.c = 1")
38:
39:     def test_stmt_assign_getitem(self):
40:         self.assertSameParse("a['b']['c'] = 1", 'a["b"]["c"] = 1')
41:
42:     def test_stmt_aug_assign_add(self):
43:         self.assertSameParse("test += 1", "test += 1")
44:
45:     def test_stmt_aug_assign_sub(self):
46:         self.assertSameParse("test -= 1", "test -= 1")
47:
48:     def test_stmt_aug_assign_mult(self):
49:         self.assertSameParse("test *= 1", "test *= 1")
50:
51:     def test_stmt_aug_assign_div(self):
52:         self.assertSameParse("test /= 1", "test /= 1")
53:
54:     def test_stmt_return(self):
55:         self.assertSameParse("return", "return")
56:
57:     def test_stmt_return_value(self):
58:         self.assertSameParse("return 1", "return 1")
59:
60:     def test_stmt_print(self):
61:         self.assertSameParse("print 1", "print 1")
62:
63:     def test_func(self):
64:         self.assertSameParse("def test(): pass",
65:                               "func test() { }")
66:

```

```

67:
68: def test_func_param(self):
69:     self.assertSameParse("def test(one): pass",
70:                           "func test(one) { }")
71:
72: def test_func_default_param(self):
73:     self.assertSameParse("def test(one=1): pass",
74:                           "func test(one=1) { }")
75:
76: def test_func_body(self):
77:     self.assertSameParse("def test(): return",
78:                           "func test() { return }")
79:
80: def test_lambda(self):
81:     self.assertSameParse("lambda x: x",
82:                           "@(x) x")
83:
84: def test_funccall(self):
85:     self.assertSameParse("test()", "test()")
86:
87: def test_funccall_param(self):
88:     self.assertSameParse("test(1)", "test(1)")
89:
90: def test_funccall_params(self):
91:     self.assertSameParse("test(1,2)", "test(1,2)")
92:
93: def test_funccall_keyword_param(self):
94:     self.assertSameParse("test(one=1)", "test(one=1)")
95:
96: def test_funccall_keyword_params(self):
97:     self.assertSameParse("test(one=1,two=2)", "test(one=1,two=2)")
98:
99: def test_cond_eq(self):
100:     self.assertSameParse("1 == 1", "1 == 1")
101:
102: def test_cond_neq(self):
103:     self.assertSameParse("1 != 2", "1 != 2")
104:
105: def test_cond_lte(self):
106:     self.assertSameParse("1 <= 2", "1 <= 2")
107:
108: def test_cond_gte(self):
109:     self.assertSameParse("2 >= 1", "2 >= 1")
110:
111: def test_cond_lt(self):
112:     self.assertSameParse("1 < 2", "1 < 2")
113:
114: def test_cond_gt(self):
115:     self.assertSameParse("2 > 1", "2 > 1")
116:
117: def test_true(self):
118:     self.assertSameParse("True", "True")
119:
120: def test_false(self):
121:     self.assertSameParse("False", "False")
122:
123: def test_and(self):
124:     self.assertSameParse("True and False", "True and False")
125:
126: def test_and_chain(self):
127:     self.assertSameParse("True and False and True", "True and False and True")
128:
129: def test_or(self):
130:     self.assertSameParse("True or False", "True or False")
131:
132: def test_or_chain(self):
133:     self.assertSameParse("True or False or True", "True or False or True")

```

```

133:
134:     def test_and_or(self):
135:         self.assertSameParse("True and False or True", "True and False or True")
136:
137:     def test_or_and(self):
138:         self.assertSameParse("True or False and True", "True or False and True")
139:
140:     def test_and_or_chain(self):
141:         self.assertSameParse("True and False or True or False", "True and False or T
rue or False")
142:
143:     def test_or_and_chain(self):
144:         self.assertSameParse("True or False and True and False", "True or False and
True and False")
145:
146:     def test_or_and_or_chain(self):
147:         self.assertSameParse("True or False and True or False", "True or False and T
rue or False")
148:
149:     def test_and_or_and_chain(self):
150:         self.assertSameParse("True and False or True and False", "True and False or
True and False")
151:
152:     def test_and_compop(self):
153:         self.assertSameParse("1 >= 2 and 3 <= 4", "1 >= 2 and 3 <= 4")
154:
155:     def test_or_compop(self):
156:         self.assertSameParse("1 >= 2 or 3 <= 4", "1 >= 2 or 3 <= 4")
157:
158:     def test_not(self):
159:         self.assertSameParse("not False", "not False")
160:
161:     def test_if(self):
162:         self.assertSameParse("if 1: pass",
163:                               "if 1 { }")
164:
165:     def test_if_cond(self):
166:         self.assertSameParse("if 1 == 1: pass",
167:                               "if 1 == 1 { }")
168:
169:     def test_if_body(self):
170:         self.assertSameParse("if 1: print 1",
171:                               "if 1 { print 1 }")
172:
173:     def test_if_else(self):
174:         self.assertSameParse("if 1:\n pass\nelse:\n pass",
175:                               "if 1 { } else { }")
176:
177:     def test_if_else_body(self):
178:         self.assertSameParse("if 1:\n print 1\nelse:\n print False",
179:                               "if 1 { print 1 } else { print False }")
180:
181:     def test_if_elseif(self):
182:         self.assertSameParse("if 1:\n pass\nelif 2:\n pass",
183:                               "if 1 { } else 2 { }")
184:
185:     def test_if_elseif_chain(self):
186:         self.assertSameParse("if 1:\n pass\nelif 2:\n pass\nelif 3:\n pass",
187:                               "if 1 { } else 2 { } else 3 { }")
188:
189:     def test_if_elseif_else(self):
190:         self.assertSameParse("if 1:\n pass\nelif 2:\n pass\nelse:\n pass",
191:                               "if 1 { } else 2 { } else { }")
192:
193:     def test_if_elseif_chain_else(self):
194:         self.assertSameParse("if 1:\n pass\nelif 2:\n pass\nelif 3:\n pass\nelse:

```

```

\n pass",
195:         "if 1 { } else 2 { } else 3 { } else { }")
196:
197:     #TODO Add ternary operator
198:     #def test_ternary(self):
199:     #    self.assertSameParse("1 if True else 2",
200:     #                          "True ? 1 : 2")
201:
202:     def test_while(self):
203:         self.assertSameParse("while 1: pass",
204:                               "while 1 { }")
205:
206:     def test_while_body(self):
207:         self.assertSameParse("while 1: print 1",
208:                               "while 1 { print 1 }")
209:
210:     def test_for(self):
211:         self.assertSameParse("for i in range(1,2): pass",
212:                               "for i := range(1,2) { }")
213:
214:     def test_for_body(self):
215:         self.assertSameParse("for i in range(1,2): print i",
216:                               "for i := range(1,2) { print i }")
217:
218:     def test_list_decl(self):
219:         self.assertSameParse("[1,2,3]", "[1,2,3]")
220:
221:     def test_property(self):
222:         self.assertSameParse("test.test", "test.test")
223:
224:     def test_getitem(self):
225:         self.assertSameParse("test[test]", "test[test]")
226:
227:     def test_binop_plus(self):
228:         self.assertSameParse("1 + 1", "1 + 1")
229:
230:     def test_binop_minus(self):
231:         self.assertSameParse("1 - 1", "1 - 1")
232:
233:     def test_binop_times(self):
234:         self.assertSameParse("1 * 1", "1 * 1")
235:
236:     def test_binop_div(self):
237:         self.assertSameParse("1 / 1", "1 / 1")
238:
239:     def test_uminus(self):
240:         self.assertSameParse("-1", "-1")
241:
242:     class ParsingBehaviorTests(unittest.TestCase):
243:         def assertSameParse(self, skit1, skit2):
244:             self.assertEqual(
245:                 ast.dump(parse_string(skit1)),
246:                 ast.dump(parse_string(skit2))
247:             )
248:
249:     def compileFunc(self, func):
250:         return parse_function(func)
251:
252:     def assertResult(self, func, result, eq=True):
253:         if eq:
254:             self.assertEqual(result, func({}))
255:         else:
256:             self.assertNotEqual(result, func({}))
257:
258:     def test_group_same_as_regular(self):
259:         self.assertSameParse("1 + 2", "(1 + 2)")

```

```
260:
261:     def test_single_double_quotes(self):
262:         self.assertSameParse("'test'", '"test"')
263:
264:     def test_range(self):
265:         self.assertSameParse("range(1,2)", "1 to 2")
266:
267:     def test_top_func(self):
268:         test = []
269:         ORACLE.set('test', test)
270:
271:         func = self.compileFunc("func(test) { return test }")
272:         test.append(1)
273:
274:         self.assertResult(func, test)
275:
276:         test.pop()
277:         self.assertResult(func, test)
278:
279:         test = [1,2,3]
280:         self.assertResult(func, test, eq=False)
281:
282:         ORACLE.set('test', test)
283:         self.assertResult(func, test)
```



```
1: from itertools import imap, chain
2: from collections import Sequence
3:
4: def listlike(obj):
5:     """Checks if the object is like a sequential container
6:
7:     Args:
8:         obj (Object): The object to check
9:
10:    Returns:
11:        Bool. True if the object is listlike, False if it's a string
12:
13:    """
14:    return isinstance(obj, Sequence) and not isinstance(obj, basestring)
15:
16:
17: def one_or_many(value):
18:     """Ensures the value can be used like a list
19:
20:    Args:
21:        value (Any): The value to check
22:
23:    Returns:
24:        Any. The value if it's listlike, or the value wrapped in a tuple if it isn't
25:
26:    """
27:    return value if listlike(value) else (value,)
28:
29:
30: def flatten(values):
31:     """Iterate over objects like a flat list
32:
33:    Args:
34:        values (List): A list of objects to flatten
35:
36:    Returns:
37:        List. A list containing the nested objects in values
38:
39:    """
40:    return chain.from_iterable(imap(one_or_many, values))
41:
42: def find_column(input, token=None, lexpos=None):
43:     """Finds the column of a token given the input it's in
44:
45:    Args:
46:        input (String) - The input being parsed
47:        token (Token) - The token being located
48:
49:    Returns:
50:        The column the token being located is in
51:
52:    """
53:    lexpos = lexpos or token.lexpos
54:    last_cr = input.rfind('\n', 0, lexpos)
55:    if last_cr < 0:
56:        last_cr = 0
57:    column = (lexpos - last_cr) + 1
58:    return column
```


./makefile

Sun May 10 18:42:49 2015

1

```
1: # makefile
2:
3: .PHONY: clean
4: clean:
5:     find . -name "*.pyc" -exec rm -rf {} \;
```