```
"""
Graph Model
"""
__author__ = 'Simon'

import networkx as nx
import graphviz
import logging
from functools import partial, reduce


logging.basicConfig(filename='example.log', level=logging.DEBUG)

draw_properties = {
    'fillcolor': {'input': '#e76f51',
                  'parameter': '#e9c46a',
                  'variable': '#f4a261',
                  'output': '#2a9d8f',
                  'computationnal': '#e76f51'},
    'fontcolor': {'input': '#eeeeee',
                  'parameter': '#eeeeee',
                  'variable': '#eeeeee',
                  'output': '#eeeeee',
                  'computationnal': '#000000'},
    'color': {'input': '#eeeeee',
              'parameter': '#eeeeee',
              'variable': '#eeeeee',
              'output': '#eeeeee',
              'computationnal': '#A9A9A9'},
    'style': {'input': 'filled',
              'parameter': 'filled',
              'variable': 'filled',
              'output': 'filled',
              'computationnal': ''},
}


class GraphModel(nx.DiGraph):
    '''GraphModel allows to write a model as a Graph.

    It herites from nx.Digraph class.

    Attributes:
        node_ordering (list): Topological order of the computationnal nodes.
    '''

    def __init__(self, graph_specifications):
        '''Initialize a graph from a specification.

        Args:
            graph_specifications(list): List of node specification.
        '''
        super(GraphModel, self).__init__()
        self.make_graph(graph_specifications)
        self.node_ordering = self.get_computational_nodes_ordering()
        self.model_function = model_function(self)
        self.graph_specifications = graph_specifications

    def checks(self, nodes, edges):
        '''Checks if the graph is well defined.

        Args:
            nodes (List): list of nodes.
            edges (List): list of edges.
        '''
        node_set = set([n[0] for n in nodes])
        edge_set = set([e[0] for e in edges])
        diff = edge_set - node_set
```

```
        assert edge_set <= node_set, f"{diff} is used in a computation but is not defined i
n a node"

    def make_graph(self, graph_nodes):
        '''Make the nx.Digraph object.

        Args:
            graph_nodes(List): list of formatted nodes

        Returns:
            None, Initialize the graph object.
        '''
        nodes, edges = GraphParser().parse(graph_nodes)
        self.checks(nodes, edges)
        self.add_nodes_from(nodes)
        self.add_edges_from(edges)
        return None

    def get_computational_nodes_ordering(self):
        '''Returns the sorted list of computationnal nodes.

        Returns:
            ordering(list): List of ordered computationnal nodes.
        '''
        ordering = [node for node in nx.topological_sort(self) if '_comp' in node]
        return ordering

    def run(self, X):
        '''Run the GraphModel given inputs and parameters.

        Args:
            X(dict): dictionnary of input and parameters.

        Returns:
            X(dict): inputs, variables and outputs of the graph.
        '''
        X = self.model_function(X)
        return X

    def draw(self, draw_properties=draw_properties):
        '''Draw the graph.

        Args:
            draw_properties(dict): dictionnary of properties for the graph plot.
        '''
        return GraphDrawer(draw_properties).draw(self)

    def draw_computation(self, inputs_parameters, draw_properties=draw_properties):
        '''Draw the graph with the computated values.

        Args:
            draw_properties(dict): dictionnary of properties for the graph plot.
            inputs(dict): dictionnary of input values.
            parameters(dict): dictionnary of parameter values.
        '''
        return GraphDrawer(draw_properties).draw_computation(self, inputs_parameters)


class GraphDrawer():
    '''A class to draw the Graph models.

    Attributes:
        draw_properties(dict): dictionnary of properties for the graph plot.
    '''

    def __init__(self, draw_properties):
        '''Initialize the GraphDrawer
```

```
        Attributes:
            draw_properties(dict): dictionnary of properties for the graph plot.
        '''
        self.draw_properties = draw_properties
        return None

    def get_node_label(self, node):
        '''Get the label of a given node.

        TO CLEAN UP !!!

        Args:
            node(node): Node of the graph.

        Returns:
            label(str): A label for the node.
        '''
        node_id, label, node_type = node[0], node[1]['name'], node[1]['type']

        if node_type != 'computationnal':
            label = f"{label} \n ({node_id})"
            if 'value' in node[1]:
                value = node[1]['value']
                label = f"{label} \n {value}"
        return label

    def draw_node(self, dot, node, draw_properties):
        '''Draw a node of the graph.

        TO CLEAN UP !!!

        Args:
            dot(dot): dot object for drawing.
            node(node): Node of the graph.
            draw_properties(dict): dictionnary of properties for the graph plot.

        Returns:
            None, updates the dot object.
        '''
        label = self.get_node_label(node)
        node_type = node[1]['type']
        dot.node(node[0], node[0], {"shape": "rectangle",
                                    "peripheries": "1",
                                    'label': label,
                                    'fillcolor': draw_properties['fillcolor'][node_type],
                                    'style': draw_properties['style'][node_type],
                                    'color': draw_properties['color'][node_type],
                                    'fontcolor': draw_properties['fontcolor'][node_type],
                                    'fontname': 'roboto'
                                    }
                 )

    def draw_edge(self, dot, a, b, draw_properties):
        '''Draw an edge of the graph.

        Args:
            dot(dot): dot object for drawing.
            a(node): Node of the graph.
            b(node): Node of the graph.
            draw_properties(dict): dictionnary of properties for the graph plot.
        '''
        dot.edge(a, b, color='#A9A9A9')

    def draw(self, G):
        '''Draw a full Graph Model

        Args:
            G(GraphModel): A graph model.
```

```
        Returns:
            dot(obj): The plot object of the graph.
        '''
        draw_properties = self.draw_properties
        dot = graphviz.Digraph(graph_attr={'splines': 'ortho'})
        for node in G.nodes(data=True):
            self.draw_node(dot, node, draw_properties)
        for a, b in G.edges:
            self.draw_edge(dot, a, b, draw_properties)
        return dot

    def draw_computation(self, G, inputs_parameters):
        '''Draw a full Graph Model with the computed values:

        Args:
            G(GraphModel): A graph model.
            inputs(dict): dictionnary of input values.
            parameters(dict): dictionnary of parameter values.

        Returns:
            dot(obj): The plot object of the graph.
        '''
        X = G.run(inputs_parameters)
        for node_id in X:
            G.nodes[node_id]['value'] = X[node_id]
        dot = self.draw(G)

        for node_id in X:  # Ugly, need to find better option for drawing
            del G.nodes[node_id]['value']

        return dot


class GraphParser():
    '''A class to parse the specification of a graph
    '''

    def __init__(self):
        '''
        Initialize a parser.
        '''
        return None

    def parse_node(self, raw_node):
        '''Parse a node.

        Args:
            raw_node(dict): raw node given in graph_specifications.

        Returns:
            node(node): A formatted non computationnal node.
        '''
        node = (raw_node['id'], {k: raw_node[k] for k in ('type', 'unit', 'name')})
        return node

    def parse_computational_node(self, raw_node):
        '''Parse a computationnal node.

        Args:
            raw_node(dict): raw node given in graph_specifications.

        Returns:
            node(node): A formatted computationnal node
        '''
        node_id = f"{raw_node['id']}_comp"
        node_param = {}
        node_param['formula'] = raw_node['computation']['formula']
```

```
        node_param['name'] = raw_node['computation']['name']
        node_param['out'] = raw_node['id']
        node_param['in'] = raw_node['in']
        node_param['type'] = 'computationnal'
        node = (node_id, node_param)
        return node

    def parse_computational_edges(self, comp_node):
        '''Parse edges from and to a computationnal node.

        Args:
            comp_node(dict): a computationnal node.

        Returns:
            edges(list): list of edges in the graph.
        '''
        edges = []
        for in_node in comp_node[1]['in']:
            edge = (in_node, comp_node[0])
            edges.append(edge)
        edges.append((comp_node[0], comp_node[0].split('_comp')[0]))
        return edges

    def parse(self, graph_specifications):
        '''Parse the graph specification

        Args:
            graph_specifications(List): list of nodes

        Returns:
            nodes(List): list of parsed nodes.
            egdes(List): list of parsed edges.
        '''
        edges, nodes = [], []
        for raw_node in graph_specifications:
            node = self.parse_node(raw_node)
            nodes.append(node)

            if 'computation' in raw_node:
                node = self.parse_computational_node(raw_node)
                nodes.append(node)

                comp_edges = self.parse_computational_edges(node)
                edges += comp_edges

        return nodes, edges


def compose(*functions):
    return reduce(lambda f, g: lambda x: f(g(X=x)), functions, lambda x: x)


def node_function(node, X):
    X = X.copy()
    function, out_node = node['formula'], node['out']
    X[out_node] = function(X)
    return X


def model_function(G):
    '''The function computed by the model'''
    functions_list = [partial(node_function, node=G.nodes[node_id])
                      for node_id in G.node_ordering[::-1]]
    return compose(*functions_list)
```