LDD2-Project: Session 1

The idea of this session is to set you up with the basic tools for creating and start manipulating graphs in Python.

Excercise1:

I. Create 2 folders: **modules/** and **tests/**
II. Create the following files:
    A. **open_digraph.py** in **modules/**
    B. **open_digraph_test.py** in **tests/**
    C. **worksheet.py** at the root
III. Test the execution from the worksheet

Excercise 2:

In **open_digraph**, define a class for the nodes and a class for the graph:

```
class node:

        def __init__(self, identity, label, parents, children):

                identity: int; its unique id in the graph
                label: string;
                parents: int->int dict; maps a parent nodes id to its multiplicity
                children: int->int dict; maps a child nodes id to its multiplicity

                self.id = identity
                self.label = label
                self.parents = parents
                self.children = children

class open_digraph: # for open directed graph

        def __init__(self, inputs, outputs, nodes):

                inputs: int list; the ids of the input nodes
                outputs: int list; the ids of the output nodes
                nodes: node iter;

                self.inputs = inputs
                self.outputs = outputs
                self.nodes = {node.id:node for node in nodes} # self.nodes: <int,node> dict
```

Excercise 3:

This exercise is aimed at creating general tests designed for troubleshooting at various stages of the project.

I. On open_digraph_test, import unittest and open_digraph:

```
import sys
import os
root = os.path.normpath(os.path.join(__file__, ))
sys.path.append(root) # allows us to fetch files from the project root
import unittest
from modules.open_digraph import *
```

II. Create a class for testing the __init__ methods of the class node:

```
class InitTest(unittest.TestCase):

        def test_init_node(self):
                n0 = node(0, 'i', {}, {1:1})
                self.assertEqual(n0.id, 0)
                self.assertEqual(n0.label, 'i')
                self.assertEqual(n0.parents, {})
                self.assertEqual(n0.children, {1:1})
                self.assertIsInstance(n0, node)

if __name__ == '__main__': # the following code is called only when
        unittest.main()          # precisely this file is run
```

III. Do the same for the class open_digraph

Exercise 4:

I. On the worksheet, import:

```
from modules.open_digraph import *
```

Create a simple graph and try to print it. What happens?

II. To obtain something more readable, implement, for both classes, the method __str__, which outputs the chain of characters used to print the graph

III. For both classes, implement the method __repr__, which "tells print what to print"

Exercise 5:

On open_digraph, implement the "class method" empty (with the @classmethod decorator), which returns an empty graph

Exercise 6:
For both classes, implement the method copy, which returns a copy of the node/graph created. On open_digraph_test, make sure that modifying a copy of an instance does not change the original instance. To do this, one can implement, for example, the test assertIsNot(x.copy( ),x)

Exercise 7:
Define and try the following "getters":

I. For the class node:
   get_id, get_label, get_parents, get_children
II. For the class open_digraph:
   A. get_input_ids, get_output_ids
   B. get_id_node_map (returns a dictionary id:node)
   C. get_nodes (returns a list of all the nodes)
   D. get_node_ids, get_node_by_id
   E. get_nodes_by_ids (returns a list of nodes given a list of ids)

Exercise 8:
Define and try the following "setters":

I. For the class node:
   set_id, set_label, set_children, add_parent_id, add_child_id
II. For the class open_digraph:
   set_inputs, set_outputs, add_input_id, add_output_id

Exercise 9:
In the worksheet, or interactively, import the inspect module and the open_digraph module. Using dir, print the list of methods corresponding to each class. Use the functions in the inspect module to find the source code for one of the methods, its doc, and the file it's in.

So far, we haven't really done any algorithmic work, but we've prepared the ground for what's to come. We should now have a functional environment, with a few files that interact properly, and we know how to communicate and interact with the data structure we've set up.

Now we'll start with something more interesting: adding nodes to a graph.

Exercise 10:
In order to add a node to the graph, we need to ensure that it is assigned an unused id. Define a new_id method which returns an unused id in the graph

Exercise 11:
Write a method add_edge(self, src, tgt) which adds an edge from the src id node to the tgt id node. Write a second method add_edges(self, edges) where edges is a list of pairs of node ids, and which adds an edge between each of these pairs

Exercise 12:
Write a method add_node(self, label='', parents=None,children=None) which adds a node (with label) to the graph (using a new id), and links it with the parent and child id nodes (with their respective multiplicities). If the default values for parents and/or children are None, assign them an empty dictionary. Return the id of the new node.

Remark: A node can have no parent, without being an input, and the same applies to outputs. Specific methods will be used to add inputs/outputs.

To have ready by next session:

I. Gather in groups of 2 or 3
II. Create a private repository per group (on GitHub/Gitlab). First, make the repository, then invite collaborators
III. Clone the repository:

```
git clone <url of repo>
```

For the future, we're only going to keep one project per group, which will be on the remote repository. When you have finished working (or even at more regular intervals), don't forget to spread your changes with:

```
git add . # or add <files>
git commit -m "<message>"
git pull
git push
```