CEng 445 Spring 2023 Projects Class Interface

The following class descriptions may not be final or not the best way to implement the projects. You can modify the arguments, add new methods as long as you can justify it.

In the projects, abbreviation CRUD implies implementation of the following methods:

Method	Description
`constructor(...)`	Create a new instance of the class from arguments
`get()`	Read. Return a textual representation of the item, you can use JSON like representation or simply records separated by punctuation (CSV)
`update(**kw)`	Update. Update the current item with new values. Updated parameters are given as keyword arguments. Other parameters remain the same
`delete()`	Delete. Delete the item.

In all projects, edited objects are maintained in global object `Repo`, a singleton catalogue object. All editable objects will be listed, added, deleted in the `Repo`:

Method	Description
`create(**kw)`	Creates a new editable object, assigns a unique id and returns the id
`list()`	Return a list (or iterator) of `(id, description)` pair for editable objects
`listattached(user)`	Lists the objects that are attached by the user
`attach(id, user)`	Returns the object with given id from `Repo`. The object is marked as *in use* as long as it is kept attached by any user.
`detach(id, user)`	The user detaches the object or stops editing and watching the object. When the last user detaches, the object will be in *not in use* state
`delete(id)`	The user deletes the object. All users should detach the object before deletion
`components`	Interface to current `Component` class

For the first phase, persistency is not required. But when implemented, the object will be loaded in the memory when the first user attaches it and saved on disk/database when the last user detaches it. Each constructed object will be assigned a unique id by `create()` and calling `getid(...)` method on the object returns it.

In all project topics there are components that composes the scene. The `Component` class is an interface for components:

Method	Description
`desc()`	Returns the description of the component
`type()`	Returns type of the component (topic specific)
`attrs()`	Returns the list, type pairs of attributes component supports
`__getattr__(attr)`	Returns the value of an attribute. If `attr` is not supported, an exception is raised
`__setattr__(attr, value)`	Sets value of an attribute. If `attr` is not supported, an exception is raised
`draw()`	Visual representation of the component instance. In the first phase, it will be a string

The following class methods are supported:

Method	Description
`list()`	List the types and descriptions of the components currently supported
`create(type)`	Construct and return an instance of a component of given type (works as a factory)
`register(type, cls)`	Register a subclass of `Component` as a new component type
`unregister(type, cls)`	Register a subclass of `Component` as a new component type

In the later phases, components can be dynamically loaded from a directory at run time.

No authentication is required in the projects. When a user starts using the library, he or she assigns `USER=value` and all calls get username from this global variable. In the second phase, the connection will get the username as the first input line. Usernames are assumed to be unique.

There will be no notification in the first phase.

Also you do not need to implement any persistency in this phase. All objects are lost when program stops.

1. Race Map Editor and Game

Main editable scene object is a `Map`.

Method	Description
`construct(cols, rows, cellsize, bgcolor)`	Constructs a map with given number of columns, rows, cell size, and background color
`__getitem__((row, col))`	Return the grid component at `(row, col)`
`__setitem__((row, col), component)`	Adds a grid component at (row, col) position in the map.
`__del__((row, col))`	Remove the grid component at the given position
`remove(component)`	Remove the component (object reference) from the map.
`getxy(y, x)`	Returns the grid component at pixel position. Grid position of a pixel is determined with modulo operations on `cellsize`
`place(obj, y, x)`	Place a player (car) component at the given pixel position
`view(y, x, height, width)`	Returns a rectangular view of the map. A view works with same interface with the `Map` but all operations are relative to `(y, x)` offset. Note that the cells at the boundaries (i.e. 0 column, 0 row) are partilly part of the view. All inspections and updates on a view affect the original map. Views of views cannot be created
`draw()`	Draw the visual representation of the race track

The map components are `Cell` objects which implement `Component` interface above. Type of cells are either `decoration`, `road`, `obstacle` or a `checkpoint`. All cells have the following attributes:

- `row`: row position on the map
- `col`: column position on the map
- `rotation`: A cell can be rotated in units of 90 degrees clock wise. `rotation` keeps number of rotations , 0 to 3.

Cells also implement (override) `interact` method described as:

- `interact(car, y, x)`: Defines a car object interacting with the cell. The details of interaction are explained below. `y` and `x` are the pixel position relative to the cell.

`Cell` components can be stacked in a grid position. Each addition is at the top and removal is from the top.

The player `Car` objects are also components with the following attributes. They are not placed as a cell of the grid but implement `Component` interface (not `Cell` interface). There can be different classes for different models.

- `model`: Model name of the car (class)
- `map`: The map the car is placed on.
- `driver`: Driver name for the current instance
- `pos`: `(y, x)` position on the map. Kept as floating point value, converted to `int` whenever necessary.
- `angle`: The direction of the car in degrees. East is 0.
- `topspeed`: Top speed of the car
- `topfuel`: Fuel capacity of the car
- `speed`: Current speed of the car. Kept as a floating point value.
- `fuel`: Current fuel level. Each move of the car reduces the fuel with the current speed. You can apply a realistic formula of your own.

When a car on the map is started, the racing starts. The following methods are defined for a `Car`.

Method	Description
`start()`	Starts the car
`stop()`	Stops the car
`accel()`	Set accelerate flag of the car as if driver presses the pedal
`break()`	Set break flag of the car as if driver presses the pedal
`left()`	Set turn left flag as player hits a left turn key
`right()`	Set turn right flag as player hits a right turn key
`tick()`	It is triggered by a game clock update event

At each `tick()` call, car internally updates its position, speed and angle based on the flags. `fuel` and `topspeed` are also relevant in the computation. `tick()` call clears the flags so that user has to set them again.

After calling `tick()` for a car, computation gets the `Cell` or cells in the cars current position. Each cell component implements `interact()` function so that cars attributes can change as:

- Speed reduces (friction)
- Speed increases (booster)
- Speed gets zero (obstacle)
- Angle randomly changes (slippery ground)
- Checkpoint recorded
- etc.

When there are stacked components on the grid node, `interact()` is called bottom to top.

Each `Cell` component type implements one or more of this behaviour. The road components reduce the speed with a small amount. Some obstacles reduce speed to 0, some only decrease it or changes the angle. Some bonus components add fuel or boost speed. When car is in a empty (no component) cell, its speed will be set to some value minimum until it goes in a road cell.

You need to implement road components for the following road types (all roads work the same way for the first phase):

- Straight, axis aligned : **━**, **┃**
- Straight, diagonal: \, /
- 90 degrees turn. **┏**, **┓**, **┛**, **┗**
- Optionally you can add 45 degrees clockwise turn, 45 degrees counter-clockwise turn, diagonal versions, crossroads etc.

In the following phases, road cells behaviour may change based on the position of the car with finer off road detection. Taking `x, y` coordinates in the consideration.

In addition to basic road components, you need to implement an obstacle component, a speed booster, and a refuel component (increasing fuel by amount). The display of a map will be simple. You can use Unicode box drawing symbols to display the grid in the text output. Output cars components in a list. The `tick()` calls will be manually triggered, just enought to test the game.

A sample test run can be given as:

```python
Repo.create("F571", 10,10, 64, 'green')
Repo.list()  # F571 will be listed with an id

ogr = Repo.attach(12345, "onur")
tgr = Repo.attach(12345, "tolga")   # these two are the same object

Repo.components.list()         # lists the available components
# assume all components call Repo.components.register(type, cls)

rt = Repo.components.create('turn90')
ogr[(1,1)] = rt
rt.rotation = 0
for j in range(2,8):
    ogr[(1,j)] = Repo.components.create('straight')
    ogr[(1,j)].rotation = 0
dt = Repo.components.create('turn90')
dt.rotation = 1
ogr[(1,8)] = dt

for i in range(2,8):
    ogr[(i,1)] = Repo.components.create('straight')
    ogr[(i,8)] = Repo.components.create('straight')
    ogr[(i,1)].rotation = 1
    ogr[(i,8)].rotation = 1

rt = Repo.components.create('turn90')
ogr[(8,1)] = rt
rt.rotation = 3
for j in range(2,8):
    ogr[(8,j)] = Repo.components.create('straight')
    ogr[(8,j)].rotation = 0
dt = Repo.components.create('turn90')
dt.rotation = 2
ogr[(8,8)] = dt

ogr[(8,3)] = Repo.components.create('booster')
ogr[(8,9)] = Repo.components.create('rock')
ogr[(8,9)] = Repo.components.create('rock')
ogr[(8,9)] = Repo.components.create('rock')
ogr[(0,8)] = Repo.components.create('rock')
ogr[(1,0)] = Repo.components.create('rock')
ogr[(7,1)] = Repo.components.create('fuel')

frr = ogr.components.create('Ferrari')
frr.driver = "Alonso"
print(frr.mode, frr.pos, frr.topspeed, frr.topfuel)
```

```
ogr.draw()
cv = ogr.view(500,500,200,200)
cv.draw()
frr.start()
frr.tick()
frr.accel()
frr.left()
frr.tick()
frr.right()
frr.accell()
frr.tick()

frr.stop()
cv.draw()
ogr.draw()
```

2. Web Dashboard

Main editable scene object is a `Dash`. A `Dash` consists of `Tab`s and each `Tab` has components placed in a 2D layout. The layout is controlled in a tabular form and components are placed on the row, column positions of the grid. Assume there are items in main menu (as in a side bar) and each tab

Method	Description
`construct(name)`	Constructs a new Dash board with name
CRUD on tabs	A dashboard will be a collection of description and `Tab` objects. The tabs can be edited as if dashboard is a dictionary.`__getitem__()`, `__delitem__`, `__setitem__` will work intuitively

A `Tab` has the following interface

Method	Description
`newrow(row=-1)`	Add a new row after `row`
`place(component, row, col=-1)`	Place a component at the given location of the layout. If column is -1, it is appended at the end of the given row
`__getitem__( (row, col) )`	Returns the component at the given position
`__delitem__( (row, col) )`	Deletes the component at the given position
`remove(component)`	Removes the component (by object references)
`view()`	It returns a visual representation of the `Tab` including all components in a layout. In the last phases this will return HTML. In the first phase a textual brief representation in a list, CSV or markdown table like format is sufficient
`refresh()`	Iterates on all components and refresh them

The components of a `Tab` are rectangular areas containing interaction and information. Consider them as simple web pages in a small rectangle. They are called `Widgets` inherited from `Component` and have the following attributes:

- `name`: Name of the component (class)
- `title`: User defined name of the component instance
- `height`: minimum height in milimeters
- `width`: minimum width in milimeters

- `env`: A dictionary of environment variable value mappings. It can be manipulated as a dictionary.
- `param`: Similar to environment variables. However parameters can be interactively updated by the user. In the final phases, parameters will be visualized as HTTP form elements
- `events`: A list of events that can be triggered on the widget. Component internally implements each event as a method. When triggered, implemented function is invoked. A `refresh` event is supported by default.
- `refresh`: Refresh interval of the component in seconds.

`Widget`s implement:

Method	Description
`view()`	Visual representation of the widget content. For the first phase, a text is sufficient
`trigger(event)`	One of the events in `events` is triggered. Corresponding call in the component implementation is called
`env`	Interface to internal `env` dictionary.
`param`	Interface to internal `param` variable value.
`refresh()`	Force refresh of the widget

In the first phase, at least the following components will be implemented:

- `URLGetter`: Get content from the `URL` attribute. Environment contains `url`.
- `MessageRotate`: Shows one of the sentences in a list of messages. Message changes each time widget is refreshed. Environment contains `messages`, a list of strings.
- `DBQuery`: Shows result of a query from an SQLite3 database. Environment contains `query`.
- `Timer`: A user controlled timer. `value` in number of seconds is given by the user as an environment variable. `pause` , `play`, `reset` events are supported. Timer is refreshed once in a second.
- `FileWatch`: A file is watched by the widget for newly appended lines. The `filename` and `numberoflines` to display is setup by an environment variable. At each refresh, file size is inspected and newly added lines are added to the model.
- `SysStat`: Shows CPU and memory usage statistics from `/proc/stat`, `/proc/loadavg` and `/proc/meminfo`. The time spend in user and system over total CPU time (including idle) gives CPU load. First 3 numbers in `loadavg` gives the load average over 1, 5 and 15 minutes. `MemAvailable` over `MemTotal` gives the memory usage. At each refresh, this numbers will be updated.
- `Chat`: A simple chat where whatever user enters (setting `param['mess']` then trigerring 'submit' event) is appended to list of lines of the component. All users of the same dash will see the same list of messages.

There will be no auto refresh in the first phase. All components refresh events will be triggered by explicit calls to widget `refresh()` or `Tab.refresh()`. Also display of a `Tab` or `Dash` does not have to be in 2D grid. Just verbal description of positions will be sufficient.

A sample test run can be given as:

```
Repo.create("Mypage")
Repo.list()  # Mypage will be listed with an id

ogr = Repo.attach(12345, "onur")
tgr = Repo.attach(12345, "tolga")   # these two are the same object

Repo.components.list()         # lists the available components
# assume all components call Repo.components.register(type, cls)

t1 = ogr.create("Personal")     # two new tabs
t2 = ogr.create("Business")
```

```
t1.newrow()
t1.newrow()

Repo.components.list()
a = Repo.components.create("URLGetter")
a.env['url'] = 'http://worldtimeapi.org/api/timezone/Europe/Istanbul'

b = Repo.components.create("MessageRotate")
b.env['messages'] = ["Live the moment.", "Work hard. Stay humble", "Be a voice, not an echo"]

c = Repo.components.create("Timer")
c.env['value'] = "30"
c.trigger('start')

d = Repo.components.create("Chat")

t1.place(a,0)
t1.place(b,0)
t1.place(c,1)
t1.place(d,1)

t1.refresh()
print(t1.view)

d.param['mess'] = "hello"
d.trigger('submit')

t1.refresh()
c.strigger('stop')
print(t1.view)
```

3. Graph Based Data Tool

The editable data tool called `Graph` is just a canvas of `Node` components. User can place nodes at a two dimensional area on screen. Then connects outgoing edges of nodes to incoming edges of other componets to form a graph. The following methods are defined for a `Graph` class.

Method	Description
`create(name, width, height)`	Create a new graph with given dimensions and name
`add(node, x, y)`	Add the component `node` at the given coordinates
`delete(node)`	Delete the component
`pick(x, y)`	Return the component at the given coordinate. Components will have a visible rectangle area. If `x, y` is inside this region, the component is returned.
`select(x, y, w, h)`	Returns the list of components in the given area defined by top left coordinates, width and height
`connect(innode, iname, outnode, oname)`	A class method to connect input port of a node to output port of the second port
`disconnect(innode, iname, outnode, oname)`	Disconnects two ports
`compute()`	Updates all nodes in the graph

All components have a set of incoming edges and outgoing edges. Nodes send tabular data through the edges. Tabular data has a set of rows, each row has a set of columns. Each column has a symbolic name and data type. You can use any internal representation of your choice. Assume this data fits in main memory.

A data `Node` component has the following attributes:

- `inputs`: a dictionary of incoming ports. Each port has a name and stores the connection information, other node and the name of the output port on the node.
- `outputs`: a dictionary of outgoing ports. Each port has a name and stores the connection information of other port.
- `params`: A dictionary of component parameters.

The `Node` components implement the following interface:

Method	Description
`indata(name)`	Returns the tabular input data available in the port with the given `name`
`outdata(name)`	Returns the tabular output data available in the port with the given `name`
`compute()`	Node makes a computation. It checks the input ports if the connected port has a newer data. If data is more recent, it makes the computation based on parameters and updates the output ports. Port data has a timestamp updated at each time computation is complete. If one of the input ports has no data or not connected, output ports are cleared

There are *importer* nodes that generates the data, from files, databases or other sources. Importers has no input ports. The *sink* components have no output ports. Data is either exported or visualized. In order to compute a graph, you can use breadth-first search or depth first search starting from importer nodes. The timestamps of port data is essential in the computation. If incoming data is newer than the last computation of the node, computation is restarted, otherwise skipped. For file importers, the timestamp of the file is used. For databases importers are always computed.

For the first phase, you need to implement:

- `CSVImport`: An importer node with a CSV file as a parameter and has an output port for the impoted data. First row of CSV file is assumed to contain column names.
- `Sorter`: Reads tabular data and sorts based on a column name given as a parameter.
- `Selector`: Reads tabular data and selects only a subset of available columns. The new table will have less columns.
- `Filter`: Reads the data and filters them based on a symbolic expression with column names and simple boolean expressions. For the first phase, you can use `lambda` expressions as parameters to define filter expressions.
- `Duplicator`: Reads the tabular data from input port and repeats it on two output ports.
- `Joiner`: Reads tabular data from two input ports and join them with a common column value. Resulting table will be a `m+n-1` column table. You can assume the join column is unique and sorted to make join more efficient.
- `Exporter`: Writes tabular data in CSV form to a file given as parameter
- `Viewer`: Prints the table on standart output. In the last phases, it will be visible as an HTML table.

A sample test run can be given as:

```
Repo.create("Finance", 800, 600)
Repo.list()  # Finance will be listed with an id

ogr = Repo.attach(12345, "onur")
```

```
tgr = Repo.attach(12345, "tolga")   # these two are the same object

Repo.components.list()         # lists the available components
# assume all components call Repo.components.register(type, cls)

c = Repo.getcomp("CSVImporter")

c.attrs()
# returns {inputs: dict, outputs: dict, params: dict, file: str}
c.file = "users.csv"

d = Repo.components.create("Exporter")
d.file = "topusers.csv"

f = Repo.components.create("Filter")
f.expr = lambda (self, grstr) : return self.getattr[grstr] > 70)


ogr.add(c, 200, 100)
ogr.add(f, 300, 100)
ogr.add(d, 400, 100)

ogr.connect(f,'input',c,'output')       # instance
Graph.connect(d,'input',f,'output')     # class call (similar)

ogr.compute()       # should import file, filter it than export
```