TinyOlap Cheat Sheet

TinyOlap is an open-source, multi-dimensional, in-memory OLAP database engine written in plain Python. As an in-process Python library, it empowers developers to build lightweight solutions for planning, forecasting, simulation, analytics and many other numerical problems. TinyOlap is also quite handy as a more comfortable alternative to Pandas DataFrames when data is multidimensional, requires many hierarchical aggregations or complex calculations.

https://github.com/Zeutschler/tinyolap
https://pypi.org/project/tinyolap

Installation

To install the latest version of a TinyOlap:

```
>> pip install tinyolap
```

To install a specific version, simply append the desired version number. It is recommended to always upgrade to the latest version of TinyOlap.

```
>> pip install tinyolap==0.8.11
```

Creating Databases

Create an empty in-memory database only. Database names should not contain blanks or special characters.

```
from tinyolap.database import Database
# create in-memory database
db = Database("tiny")
```

To create a persistent database, you either specify a valid path incl. a database name (missing folders will be created automatically) or you need to set the optional `in_memory` argument to `False`. If no path is defined, then the database will be saved in a folder `/db/` aside the calling Python script.

```
from tinyolap.database import Database
# create persistent databases
db1 = Database("/user/…/tiny1.db")
db2 = Database("tiny2", in_memory=False)
```

At any time, you can create a snapshot of an in-memory or persistent database using the `export(…)` method. Again, if no path is defined, then the folder `/db/`will the output dir.

```
db.export("/user/…/other.db") # create snapshot
```

Creating Dimensions

Before you can create a cube, you need to create a set of dimensions that define the dataspace for the cube. To add members to a dimension, you need set the dimension into `edit()` mode. Then you can add or manipulate members. Finally, you need to `commit()` your changes. The database will then perform a short reorganization as a change of dimension members may impact existing data in cubes, so deleting a dimension member will delete all data in all cubes related to that dimension member.

The following code creates a dimension with 3 members.

```
years = db.add_dimension("years")
years.edit()
years.add_member("2021")
years.add_member("2022")
years.add_member("2023")
years.commit()
```

You can also add a list of members using the same method. In addition, TinyOlap supports method chaining.

```
regions = db.add_dimension("regions").edit()
   .add_member(
       ["North", "South", "West", "East"]
   ).commit()
```

Finally, Dimension can be converted into or created by json using the methods `to_json(…)` and `from_json(…)`.

Member Hierarchies

Hierarchies are essential for multi-dimensional data aggregation. TinyOlap supports unbalanced as well as redundant aggregations within one hierarchy, meaning each member can aggregate into multiple parent members.

```
regions.edit().add_member(
       member="NE",
       children=["North", "East"])
   .commit()
```

This is also supported for adding a list of members but requires the children to be in separate lists or tuples. Child members that do not exists will be created. A full example:

```
months = db.add_dimension("months").edit()
   .add_member(
       ["Q1", "Q2", "Q3", "Q4"],
       [("Jan", "Feb", "Mar"),
        ("Apr", "Mai", "Jun"),
        ("Jul", "Aug", "Sep"),
        ("Oct", "Nov", "Dec")])
    .add_member(
       "Year"
       ["Q1", "Q2", "Q3", "Q4"],
    .add_member(
       "Summertime"
       ["Jun", "Q3"],) # unbalanced & multiple
    .commit()
```

Member Subsets

Subsets are flat lists of members. Subsets are useful for reporting purposes as well as custom aggregations. Adding subsets does not require the dimension to be in edit mode.

```
dim.add_subset("name", [list of members])
```

Member Attributes

Attributes are properties of the members in a dimension. If a data type is defined, then setting or changing attribute values will be type checked. Attributes are very useful for reporting, data integration as well as custom aggregations. Adding attributes does not require the edit mode.

```
dim.add_attribute("desc", str)
dim.add_attribute("date", datetime.date)
dim.add_attribute("anything")
dim.add_attribute("special", MyObject)
```

Reading and writing attributes can be done through the dimension or (more convenient) the member object:

```
months.set_attribute("Jan", "weather", "bad")
months.add_attribute("Aug", "avg. temp", 27.3)
jan = months["Jan"]
jan["weather"] = "very bad"
```

Creating Cubes

Cubes define and span data space through a combination on 1 to N dimensions. Cube creation is straight forward:

```
cube = db.add_cube(name="data",
     dimensions=["years", "months", "regions"])
```

Creating Rules

To define advanced business logic, TinyOlap provides the concept of rules. Rules are normal Python functions or (static) class methods but need to be annotated as shown. Once defined, rules need to be registered for the cube.

```
@rule("data", ["2023"])
def rule_future(self, c: Cell):
    return c["2022"] * 2.0

cube.register_rule(rule_future)
```

Reading and Writing Data

Data access is provided through cells and data areas. Cell access requires to define a member name for each of the dimensions of a cube.

```
cube["2022", "Jan", "North"] = 123.0
cube["2022", "Feb", "East"] = 234.0
v = cube["2022", "Q1", "NE"] # will return 357.0
```

A more convenient way to access data is provided through the `Cell` object. Comparable to cursors in a relational DB. Cell objects can be used in mathematical operations.

```
c = cube.cell("2022", "Feb", "East")
c. Value = 124.0
c.value = c * 2 # will writes 248.0 to the cube
```

Cell objects can be easily 'bend' to point towards other cells and used to navigate the multi-dimensional data space.

```
c["Mar"] = 100.0 # = c["2022", "Mar", "East"]
c["months:Mar"] = 200.0 # for ambiguous members
```

Data Areas

Data areas are a very powerful feature. They allow the access and modification of areas (sub-spaces) in a cube. Otherwise, they behave like cells and support mathematical operations.

```
a = cube.area()  # represents the entire cube
a["2022"] *= 2.0 # all 2022 data get multiplied
print(a.max())   # will print 496.0

a["Feb"] = a["Jan"] * 3.0 # over * years/regions

# clears the destination, then copies all data
a["Jan", "2022"] = a["Jun", "2021"]
```

SQL Queries

TinyOlap has a rudimentary support for SQL dimensions, members, attributes, subset and cubes.

```
sql = "SELECT * FROM data WHERE '2022'"
records = Query(db, sql).execute().records()
```

To explore all capabilities of TinyOlap, please visit tinyolap.com