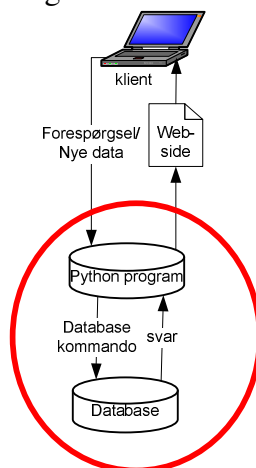


Øvelse 9. Klasser, objekter og sql-tabeller

Denne opgave handler om hvordan man opbevarer data fra databasekald på en struktureret måde. Den skal samtidig give jer erfaringer med objekter, der kommer til at spille hovedrollen i systemudvikling til foråret.



Når man indlæser data fra en database til sit program, skal man finde en hensigtsmæssig datastruktur hvori man kan opbevare data. I dette kursus skal I bruge objekter, for det kommer til at spille hovedrollen i systemudvikling til foråret.

Opgaven ligger i forlængelse af opgave 6 og 7 hvor I lærte at kommunikere mellem python og databasen. Denne opgave handler om hvordan man opbevarer de oplysninger man får tilbage og om hvordan man organiserer python-programmet på en velstruktureret måde. I denne opgave vil resultater af databasekald blive opbevaret i objekter.

Den ligger også i forlængelse af opgave 5 fordi klassen SQL skal have en funktion, *makeHTMLTable*, der kan fremvise dens data som en html-tabel.

Opgaven går ud på at gøre nedenstående modul færdige, dvs. indsætte kode alle de steder hvor der står '**insert code here**'. Nogle steder er angivet hvilket databasekald man skal benytte sig af. Modulet beskriver klasser hvis instanser kan bruges til at gemme sine data i.

Modulets klasser er beskrevet i fig 1.

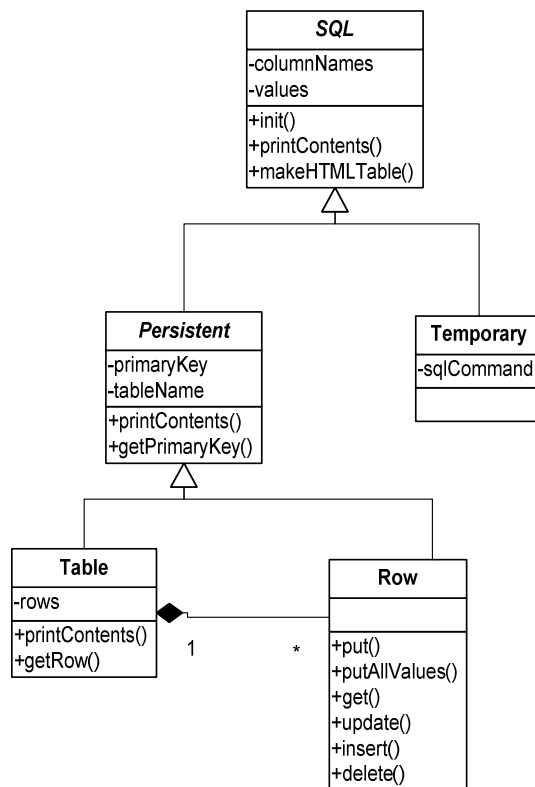


Fig. 1.

Klasserne *SQL* og *Persistent* er abstrakte klasser, dvs. de er ikke beregnet til at man skal lave instanser af dem.

SQL klassen indeholder to attributter: *columnNames* er en liste af navne på databasekolonner og *values* der er en liste af lister der repræsenterer rækker i tabel. Desuden indeholder den en funktion *makeHTMLTable()* der producerer en html-tabel af sine data. Det var den I lavede i øvelse 6. Da *SQL* er overklasse til alle andre klasser, vil disse nedarve *columnNames*, *values* og *makeHTMLTable()*.

Persistent repræsenterer tabeller og tabellerrækker i databasen. Det er en betingelse at de har en primary key.

Table repræsenterer en tabel og *Row* repræsenterer en række.

Hvis *Row* genereres med en nøgle (barn = Row('Child','3112450091')), vil instansen indeholde data fra posten med den pågældende nøgle. Man kan så ændre data med *put*-metoden (barn.put('firstname','Jeppe')), og senere gemme resultat ved hjælp af *update*-metoden (barn.update()). Man kan også slette rækken i databasen ved hjælp af *delete* (barn.delete()). Genereres *Row* uden nøgle, oprettes en tom instans som man kan fylde indhold på ved hjælp af *put*-metoden. I starten er *values* = ["", "", "", "", "", "", "", "", "", ""]. Man kan også indsætte alle data på én gang ved hjælp af *putAllValues()*:

```

valueList = ['1111111111', 'Jeppe Boegh', 'Andersen', 'Thorsgade 20', '8410',
'dreng', '', '86379790', 'Baltica', '1111111112', '1111111113', '1111111114']
x.putAllValues(valueList)
  
```

Instansen kan så senere indsættes som en ny post i databasen ved hjælp af *insert*-metoden (barn.insert()).

Table indeholder ud over *values* (den liste af liste af tabelrækker som alle klasser indeholder) en liste *rows* der indeholder Row-objekter svarende til de rækker der forekommer i *values*.

Temporary-klassen er beregnet til at gemme vilkårlige databaseudtræk. Den genereres med en liste af kolonnenavne og en SQL-ordre. Kolonnenavne skal svare til de kolonnenavne der nævnes efter SELECT i SQL-ordren. Altså f.eks.:

```
SQLordre = '''SELECT Child.cpr, Child.firstname, Child.lastname FROM Child, Person
WHERE Child.hasMother = Person.cpr AND Child.address = Person.address'''
barn = Temporary(['cpr', 'firstname', 'lastname'], SQLordre)
```

SQL-ordren finder cpr, fornavn og efternavn på alle børn der bor på moderens adresse. Resultatet er i dette tilfælde én person,

```
[['1111111115', 'Stine', 'Jacobsen ']]
```

Også her kan vi få lavet en html-tabel ved at skrive

```
barn.makeHTMLTable()
```

Det giver følgende tabel:

cpr	firstname	lastname
1111111115	Stine	Jacobsen

Nedenfor er den kode I skal færdiggøre. I kan downloade den fra hjemmesiden. Der hedder den *OOogSql*. Husk at lægge modulet *database.py* (som I downloadede i øvelse 6) i samme mappe, for *OOogSql* importerer *database.py*

```
import database

class SQL:
    '''Abstract class
    columnnames is a list of column names
    values is a list of lists representing rows in the database'''
    def __init__(self, columnNames, values):
        self.columnNames = columnNames
        self.values = values
    def printContents(self):
        '''prints the data of the object'''
        print 'Column names: '+str(self.columnNames)
        print 'Values: '+str(self.values)
    def makeHTMLTable(self):
        '''makes a html-table that displays the column names and the values'''
        tableWithHeading = []
        tableWithHeading.append(self.columnNames)
        tableWithHeading.extend(self.values)
        return makeTable(tableWithHeading)

class Persistent(SQL):
    '''Abstract class'''
    def __init__(self, tableName, values):
        self.primaryKey = database.getPrimaryKey(tableName)
        self.tableName = tableName
        columnNames = database.getColumnNames(tableName)
        SQL.__init__(self, columnNames, values)
    def printContents(self):
        '''prints the data of the object'''
        print 'Table name: '+self.tableName
        print 'Primary key: ' + self.primaryKey
        SQL.printContents(self)
    def getPrimaryKey(self):
        '''returns the primary key'''
        #insert code here
```

```

class Table(Persistent):
    ''' This class represents a table
    Values contains a list of lists representing the rows in the table.
    Components contains corresponding instances of the Row class
    Instances are generated by Table(tablename)'''
    def __init__(self, tableName):
        values = database.findRecords(tableName, [])
        valuelist = []
        Persistent.__init__(self, tableName, values)
        #insert a list of Row-instances corresponding to the values
        rows = []
        primaryKey = self.primaryKey
        i = self.columnNames.index(primaryKey)
        for v in self.values:
            theKey = v[i]
            rows.append(Row(tableName, theKey))
        self.rows = rows
    def getRow(self, theKey):
        '''returns a row-instance whose primary key = theKey
        if none exists, returns None'''
        #insert code here

    def printContents(self):
        '''prints the data of the object'''
        print 'Table name: '+self.tableName
        print 'Primary key: '+ self.primaryKey
        print 'Values : \n' + str(self.values)
        print 'Components: '
        for c in self.rows:
            c.printContents()

class Row(Persistent):
    ''' This class represents a row in a table.
    The data of the row is represented by a list containing one list:
    [[data, data, data, data]]
    If a key is provided, the row in the database with the key
    is read into the instance.
    If no key is provided, an instance with empty values is generated
    Instances are generated by Row(tablename) or Row(tablename, key)'''
    def __init__(self, tableName, key=''):
        Persistent.__init__(self, tableName, None)
        if key != '':
            #insert the table row
            self.values = database.findRecords(tableName, [], {self.primaryKey:key})
        else:
            #insert an empty table row
            theLength = len(self.columnNames)
            self.values = ['']*theLength
    def put(self, attribute, value):
        ''' inserts value as the value of the attribute
        If the attribute does not exist, it prints attribute does not exist'''
        #insert code here

    def putAllValues(self, valueList):
        '''inserts valueList as the value of the instance'''
        if len(valueList) == len(self.columnNames):
            self.values = valueList
        else:
            print 'length of valueList is different from length of columnNames'
    def get(self, attribute):
        '''returns the value of attribute if it exists.
        Else return None'''
        #insert code here

    def update(self):
        '''updates the database'''
        theKey = self.get(self.primaryKey)
        if theKey != '':
            updateList = database.rowToDictionary(self.columnNames, self.values[0])
            database.updateRecords(self.tableName, updateList,
            {self.primaryKey:theKey})
        else:
            print 'cannot update because the key is missing'
    def insert(self):
        '''inserts the row as a new row in the database if
        the primary key exists. Else print
        "cannot insert because key is missing"'''
        #insert code here

    def delete(self):
        '''deletes the row from the database if the primary key exists
        Else print "cannot delete because the key is missing"'''
        #insert code here

class Temporary(SQL):

```

```
'''Class representing an arbitrary selection from the database
Instances are generated by Temporary(columnNames, sqlCommand)
where columnNames is a list of column names'''
def __init__(self, columnNames, sqlCommand):
    SQL.__init__(self, columnNames, database.queryDB(sqlCommand))

def makeTable(tablecontents):
    ''' Produces a html table.
    Input is a list of list [a1,a2,a3...] where ai is a list.
    Each ai is visualised as a row in the html-table.
    '''
    #insert the function you made in exercise 6 here
    return htmltable
```