Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

MCS 275 Lecture 26
Programming Tools and File Management
Jan Verschelde, 10 March 2017

Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

a GUI to our Database of Python Scripts

We made one table `scripts` in the database `OurPyFiles`, storing data about our Python scripts.

For every Python script we have 4 fields:
its type, number, date, and file name.
Example of a header:

```
# L-26 MCS 275 Fri 10 Mar 2017 : guidb1.py
```

Corresponding data tuple:

```
('L', '26', '2017-03-10', 'guidb1.py')
```

With a GUI we get a better overview.
We will sort the records in various ways.

Widgets in the GUI

Message field to write status information.

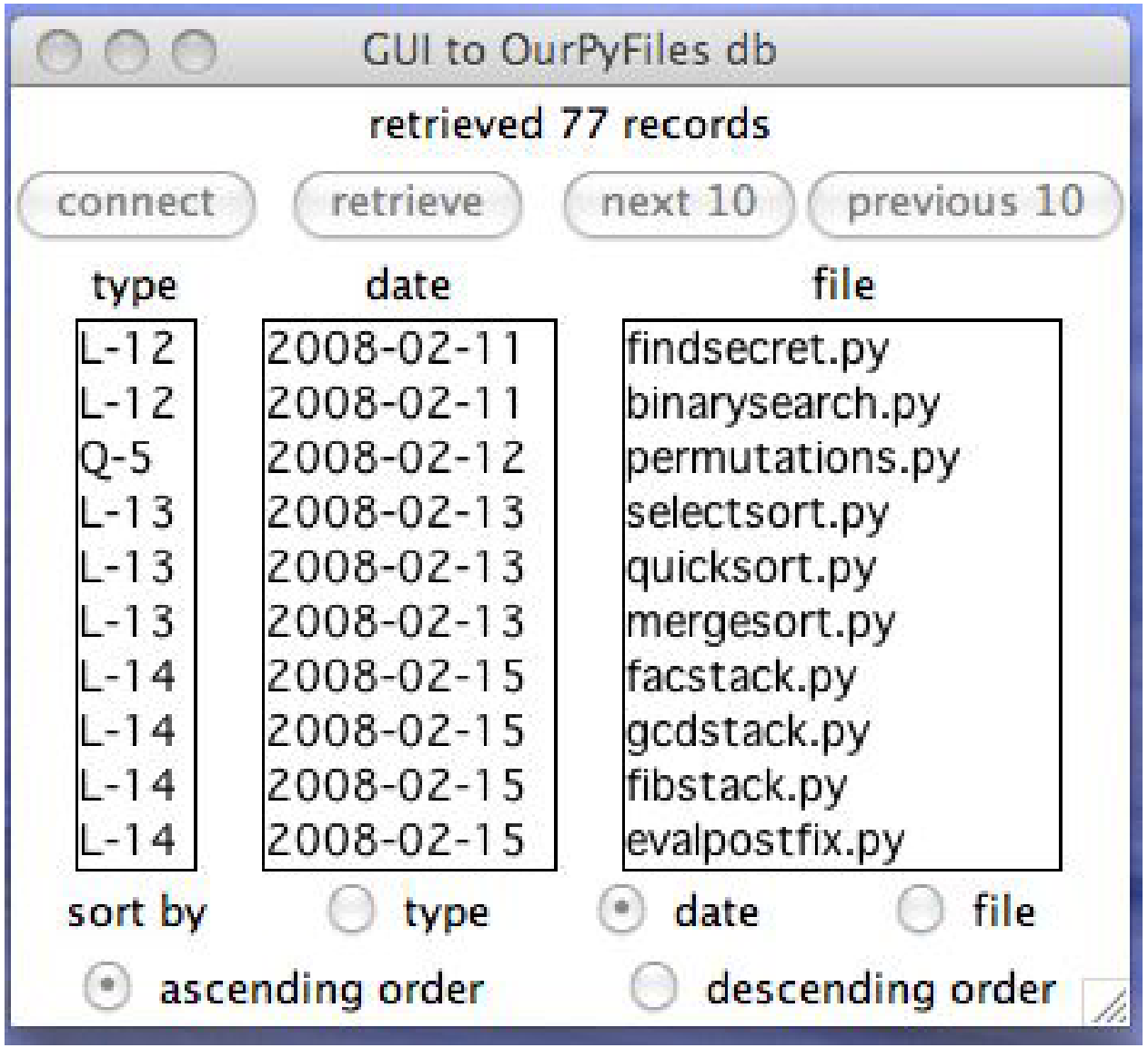
Buttons to . . .

- connect to the database
- retrieve all records
- see next 10 or previous 10 records

Radio buttons to determine

- which field to use for sorting
- sort in ascending or descending order

viewing the data in the table scripts



Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

message label and other labels

Code in `__init__`:

```
self.message = StringVar()
self.message.set("welcome to our database")
self.messageLabel = Label(wdw, \
    textvariable = self.message)
self.messageLabel.grid(row=0, column=0, columnspan=4)

self.tt = Label(wdw, text='type')
self.tt.grid(row=2, column=0)
self.dd = Label(wdw, text='date')
self.dd.grid(row=2, column=1)
self.ff = Label(wdw, text='file')
self.ff.grid(row=2, column=2, columnspan=2)

self.sL = Label(wdw, text='sort by')
self.sL.grid(row=4, column=0)
```

connecting to the database

Button defined in `__init__`:

```
    self.bc = Button(wdw, text='connect', \
       command=self.connect)
    self.bc.grid(row=1, column=0)
    self.cursor = 0

def connect(self):
    """
    Connects to the database OurPyFiles.
    """
    try:
        # db = MySQLdb.connect(db="OurPyFiles")
        db = pymysql.connect(db="OurPyFiles")
        self.message.set("connected to \"OurPyFiles\"")
        self.cursor = db.cursor()
    except:
        self.message.set("failed to connect to \"OurPyFiles\"")
```

radio button to sort

We can sort on `type`, `date`, or `file`.

Code for Radiobutton in `__init__`:

```
self.RadioSort = IntVar()
self.st = Radiobutton(wdw, text='type', \
    variable=self.RadioSort, value = 1)
self.st.grid(row=4, column=1)
self.sd = Radiobutton(wdw, text='date', \
    variable=self.RadioSort, value = 2)
self.sd.grid(row=4, column=2)
self.sf = Radiobutton(wdw, text='file', \
    variable=self.RadioSort, value = 3)
self.sf.grid(row=4, column=3)
self.RadioSort.set(1)
```

The default sort is set to `type`.

radio button to order

User has choice between ascending or descending order.

Code for Radiobutton in `__init__`:

```
self.RadioOrder = IntVar()
self.asc = Radiobutton(wdw, text='ascending order', \
    variable = self.RadioOrder, value = 1)
self.asc.grid(row=5, column=0, columnspan=2)
self.dsc = Radiobutton(wdw, text='descending order', \
    variable = self.RadioOrder, value = 2)
self.dsc.grid(row=5, column=2, columnspan=2)
self.RadioOrder.set(1)
```

The default order is set to `ascending`.

formulating the query

```
def query(self):
    """
    Returns the query to select all records,
    taking input from the radio buttons.
    """
    q = 'select * from scripts'
    ord = ''
    if self.RadioOrder.get() == 1:
        ord = 'asc'
    elif self.RadioOrder.get() == 2:
        ord = 'desc'
    if self.RadioSort.get() == 1:
        q = q + ' order by t, n ' + ord
    elif self.RadioSort.get() == 2:
        q = q + ' order by d ' + ord
    elif self.RadioSort.get() == 2:
        q = q + ' order by f ' + ord
    return q
```

Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

retrieving records – command for the second button

```
def retrieve(self):
    """
    Retrieves all records from the scripts table.
    """
    if self.cursor == 0:
        self.message.set("please connect first")
    else:
        q = self.query()
        lc = self.cursor.execute(q)
        m = 'retrieved %d records' % int(lc)
        self.message.set(m)
        R = self.cursor.fetchall()
        self.clear()
        for i in range(len(R)):
            if i >= 10:
                break
            self.insert(R[i])
        self.records = R
        self.pos = 10
```

displaying data in listboxes

To display the data, we use Listbox widgets.

We will use three listboxes (defined in `__init__`):

1. for type and number, e.g.: `L-26`

```
self.Lt = Listbox(wdw, width=4, height=10)
self.Lt.grid(row=3, column=0)
```

2. for the date, e.g.: `2017-03-10`

```
self.Ld = Listbox(wdw, width=10, height=10)
self.Ld.grid(row=3, column=1)
```

3. for the file name, e.g.: `guidb1.py`

```
self.Ln = Listbox(wdw, width=15, height=10)
self.Ln.grid(row=3, column=2, columnspan=2)
```

using listboxes

```
def clear(self):
    """
    Clears all listboxes.
    """
    self.Lt.delete(0, END)
    self.Ld.delete(0, END)
    self.Ln.delete(0, END)

def insert(self, item):
    """
    Inserts one record item to the listboxes.
    """
    t = item[0] + '-' + str(int(item[1]))
    self.Lt.insert(END, t)
    self.Ld.insert(END, str(item[2]))
    self.Ln.insert(END, item[3])
```

buttons to navigate

Because our listboxes are only 10 in height,
we cannot display everything all at once.

Two navigation buttons in `__init__`:

1. to show the next 10 records:

```
self.bn = Button(wdw, text='next 10', \
    command=self.next10)
self.bn.grid(row=1, column=2)
```

2. to show the previous 10 records:

```
self.bp = Button(wdw, text='previous 10', \
    command=self.prev10)
self.bp.grid(row=1, column=3)
```

Storing the current position in the retrieved records:

```
self.pos = 0
```

displaying the next 10 records

```
def next10(self):
    """
    Displays the next 10 records in listboxes.
    """
    if self.records == 0:
        self.message.set("no records to show")
    else:
        self.clear()
        for i in range(self.pos, self.pos+10):
            if i >= len(self.records):
                break
            self.insert(self.records[i])
        self.pos = self.pos + 10
        if self.pos >= len(self.records):
            self.pos = len(self.records) - 1
```

displaying the previous 10 records

```
def prev10(self):
    """
    Displays the previous 10 records in listboxes.
    """
    if self.records == 0:
        self.message.set("no records to show")
    else:
        self.clear()
        self.pos = self.pos - 20
        if self.pos < 0:
            self.pos = 0
        for i in range(self.pos, self.pos+10):
            if i >= len(self.records):
                break
            self.insert(self.records[i])
        self.pos = self.pos + 10
```

Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

Making the Database more useful

splitting the table scripts

The four fields were copied from the headers.

Two drawbacks:

1. only the file name is unique for every record
   and not so convenient to use as key
2. as most lectures featured more than one script,
   there is a lot of redundant data in the table

We normalize, using scripts to create two new tables:
one with file names and one with type and dates.

Because tedious to do manually with the mysql monitor,
we develop a script.

the function `main()`

```
def main():
    """
    Splits scripts in two tables.
    """
    # db = MySQLdb.connect(db="OurPyFiles")
    ourdb = pymysql.connect(db="OurPyFiles")
    crs = ourdb.cursor()
    qry = 'select * from scripts order by d'
    nbr = crs.execute(qry)
    print('found %d records' % int(nbr))
    ans = input('Make the tables ? (y/n) ')
    if ans == 'y':
        make_tables(crs)
```

`main()` continued

```
data = crs.fetchall()
(types, files) = split_records(data)
print('T = ', types)
print('F = ', files)
if ans == 'y':
    ans = input('Do the inserts ? (y/n) ')
    insert_type_date(crs, types, ans=='y')
    insert_file_data(crs, files, ans=='y')
    date_files(crs)
    ourdb.commit()
else:
    print('showing the insert queries')
    insert_type_date(crs, types)
    insert_file_data(crs, files)
```

making the tables

```
def make_tables(crs):
    """
    Executes the MySQL commands to create
    the tables typedate and filedata.
    The input parameter crs is the cursor.
    """
    make_type_date(crs)
    make_file_data(crs)
```

make table typedate

```
def make_type_date(crs):
    """
    The table typedate stores type and date,
    e.g.: L-26 and 2017-03-10, repesented by
    four fields: id, type, number, and date.
    The input parameter crs is the cursor.
    """
    try:
        qry = 'create table typedate ' + \
          '( i INT, t CHAR(1), n INT, d DATE )'
        crs.execute(qry)
        print(qry + ' succeeded')
    except:
        print(qry + ' went wrong...')
```

make table filedata

```
def make_file_data(crs):
    """
    The table filedata stores file names,
    represented by three fields: id, name,
    and id of the corresponding entry in
    the typedate table, crs is the cursor.
    """
    try:
        qry = 'create table filedata ' + \
          '( i INT, f CHAR(20), t INT )'
        crs.execute(qry)
        print(qry + ' succeeded')
    except:
        print(qry + ' went wrong...')
```

Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

splitting records

```
def split_records(R):
    """
    Returns two lists: records that go
    in typedate and in filedata.
    """
    L = []; T = []; F = []; cnt = -1
    for i in range(len(R)):
        d = R[i]
        p = (d[0], d[1], d[2])
        if p in L:
            ind = L.index(p)
        else:
            cnt = cnt + 1
            L.append((d[0], d[1], d[2]))
            T.append((cnt, d[0], d[1], d[2]))
            ind = cnt
        F.append((i, d[3], ind))
    return (T, F)
```

inserting type and date

```
def insert_type_date(crs, T, doit=False):
    """
    Given the cursor and a list of values for
    the table typedate, all records are added.
    """
    for i in range(len(T)):
        p = T[i]
        qry = 'insert into typedate values ' + \
            '(\"%s\", \"%s\", \"%s\", \"%s\") ' \
            % (str(p[0]), str(p[1]), str(p[2]), \
               str(p[3]))
        if doit:
            crs.execute(qry)
        else:
            print('query = ', qry)
```

inserting file names

```
def insert_file_data(crs, F, doit=False):
    """
    Given the cursor and a list of values for
    the table filedata, all records are added.
    """
    for i in range(len(F)):
        n = F[i]
        qry = 'insert into filedata values ' + \
            '(\"%s\", \"%s\", \"%s\") ' \
          % (str(n[0]), n[1], str(n[2]))
        if doit:
            crs.execute(qry)
        else:
            print('query = ', qry)
```

a sanity check

```
def date_files(crs):
    """
    As a final check, selects file name
    and corresonding date from the newly
    created tables, crs is the cursor.
    """
    qry = 'select f,d from filedata, typedate' \
        + ' where filedata.t = typedate.i'
    crs.execute(qry)
    res = crs.fetchall()
    print(res)
    print('#records :', len(res))
```

Interfacing with MySQL

1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

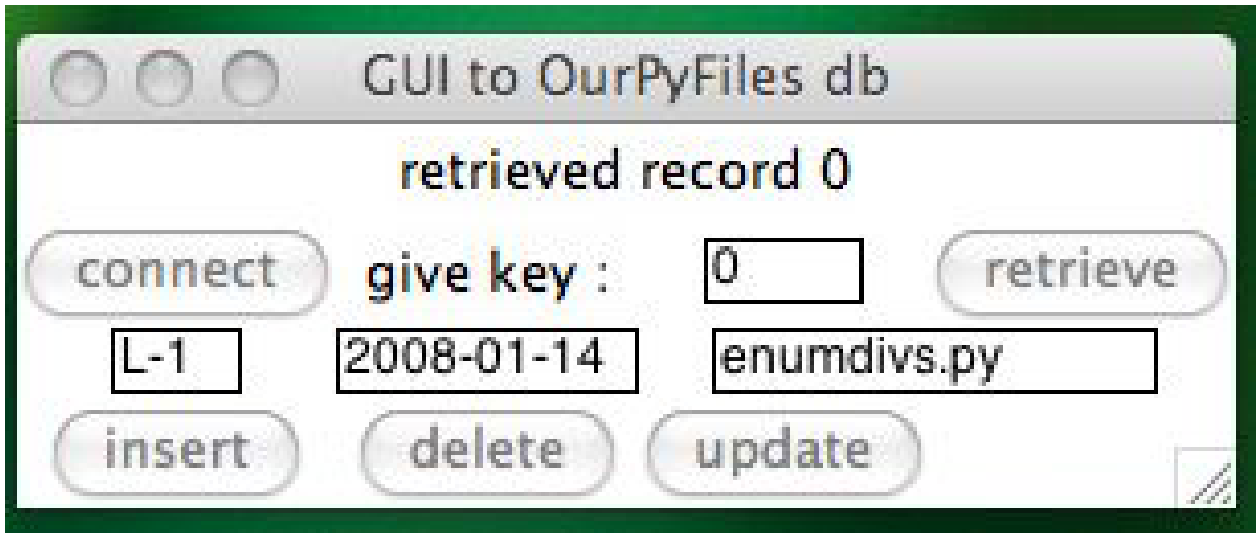
Using Keys

After the normalization, every script has a unique key.
This key is in the `i` field of the table `filedata`.

Accessing the data via the key allows us to

1. select records directly
2. insert, delete, and update records.

The Widgets in the GUI



Query for Specific Record

```
def query(self, key):
    """
    Returns the query for all the information
    of the script with the given key.
    """
    q = 'select typedate.t, n, d, f ' + \
        'from typedate, filedata ' + \
        'where filedata.t = typedate.i ' + \
        'and filedata.i = %d' % key
    return q
```

Interfacing with MySQL

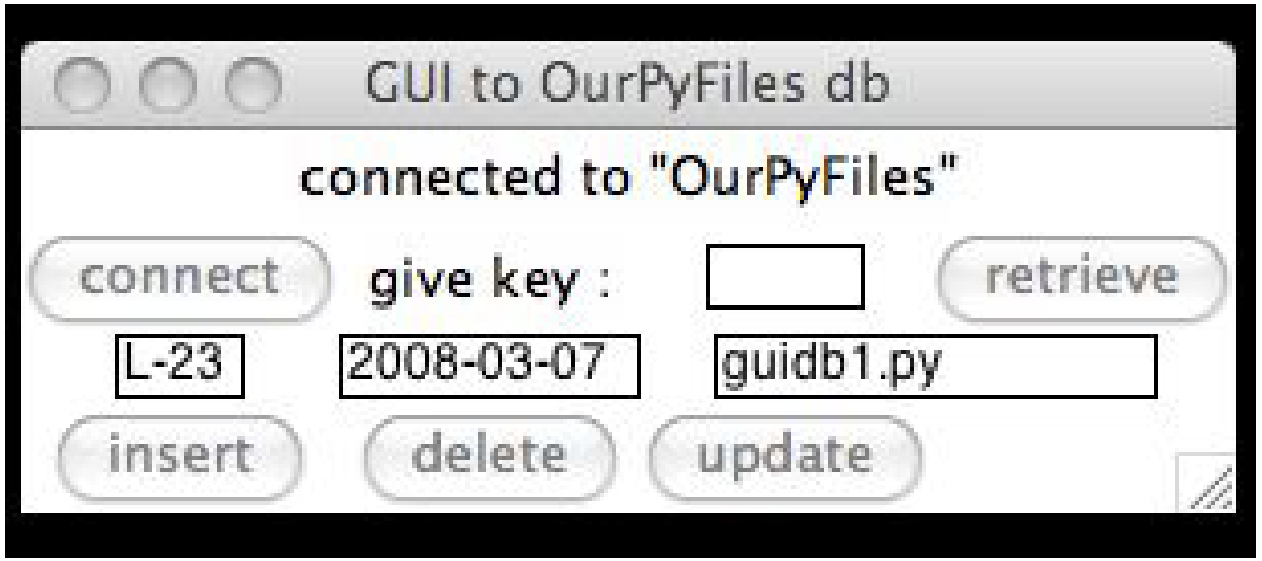
1. A GUI to browse a MySQL table
   - our database with Python scripts
   - connecting to database, setting sort order
   - retrieving and displaying records
2. Normalization
   - splitting a table in two
   - moving data into the tables
3. Retrieving Specific Records
   - use the key to access a record
   - inserting a new record in a table

counting the number of files

```
def count_files(self):
    """
    Returns the number of files.
    """
    qry = 'select count(*) from filedata'
    ret = self.cursor.execute(qry)
    nbr = self.cursor.fetchone()
    return int(nbr[0])
```

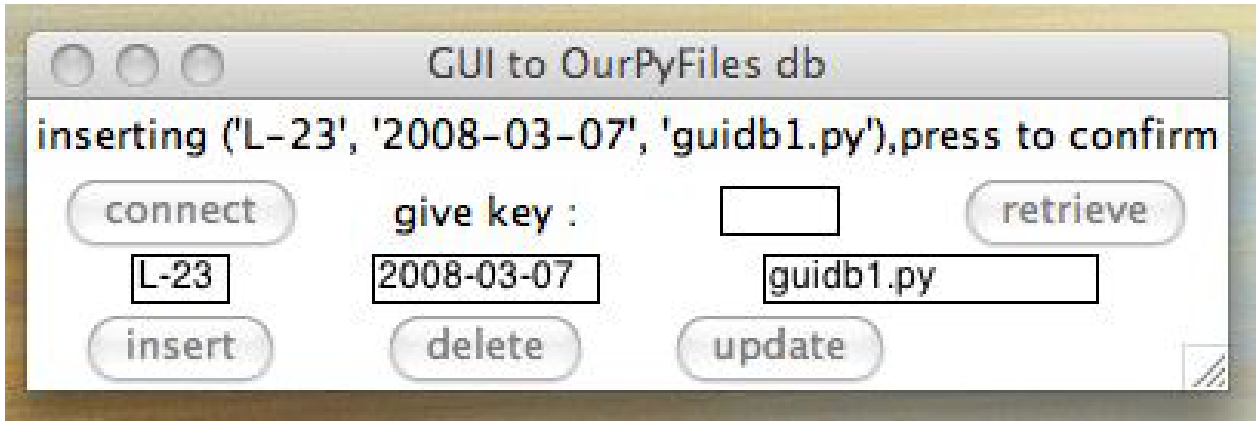
inserting a new file

Entering the data:



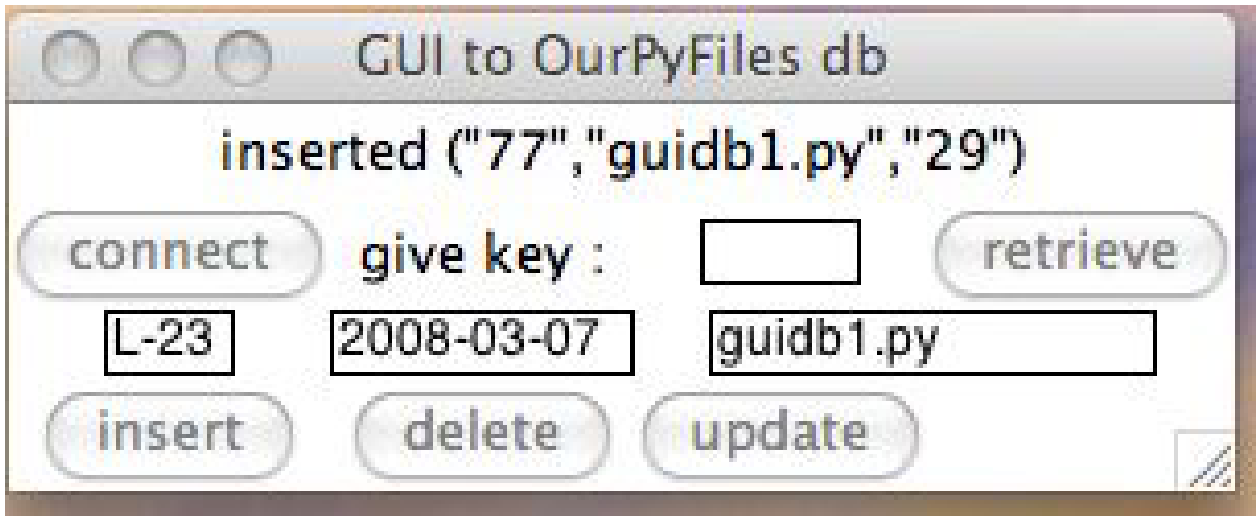
inserting a new file

Asking to confirm the data:



inserting a new file

Confirming the insert:



inserting data

```
def insert_data(self):
    """
    Inserts a new record into the database.
    """
    t = self.tp.get()
    d = self.dt.get()
    f = self.nm.get()
    r = (t, d, f)
    if not self.startinsert:
        m = 'inserting %s,' % str(r)
        m = m + 'press to confirm'
        self.message.set(m)
        self.startinsert = True
    else:
        self.insert_record(r)
        m = 'inserted %s' % str(r)
        self.startinsert = False
```

inserting a record

```
def insert_record(self, data):
    """
    Prepares data to insert a record.
    """
    L = data[0].split('-')
    (t, n) = (L[0], L[1])
    (d, f) = (data[1], data[2])
    nf = self.count_files()
    nt = self.count_types()
    v = '(' + '\"' + str(nf) + '\"' + \
        ',' + '\"' + f + '\"' + ',' + \
        '\"' + str(nt) + '\"' + ')'
    self.insert_file_values(v)
    m = 'inserted %s' % v
    self.message.set(m)
```

inserting values

Recall the MySQL command:

```
insert into <table> values <tuple>
```

```
def insert_file_values(self, vals):
    """
    Inserts values in filedata table.
    """
    qry = 'insert into filedata values '
    qry = qry + vals
    self.cursor.execute(qry)
```

Summary and Assignments

A good reference on GUIs with Tkinter:
`http://infohost.nmt.edu/tcc/help/pubs/tkinter.pdf`
visit `http://www.mysqltutorial.org/`

Exercises:

1. Create the equivalent to `guidb1.py`,
   our first GUI, writing a CGI script.
2. The method in our second GUI to insert is not yet complete.
   Also provide functions to add the corresponding values in the table `typedate`.
3. Provide functions to delete records.
4. Provide functions to update records.