KirbyBase Manual

Version 1.9

By Jamey Cribbs
jcribbs@twmi.rr.com

Introduction

KirbyBase is a simple, pure-Python, flat-file database management system. Some of its features include:

- Since KirbyBase is written in Python, it runs anywhere that Python runs.
- All data is kept in plain-text, delimited files that can be edited by hand. This gives you the ability to make changes by just opening the file up in a text editor, or you can use another programming language to read the file in and do things with it.
- It can be used as an embedded database or in a client/server, multi-user mode. To switch from one mode to the other, you only have to change one line in your program. Included in the distribution ar two sample database server scripts. One is single-threaded, blocking, and pretty safe. The other one is a multi-threaded, non-blocking script that has worked well in limited testing.
- Tables are kept on disk during use and accessed from disk when selecting, updating, inserting, and deleting records. Changes to a table are written immediately to disk. KirbyBase is not an "in-memory" database. Once you update the database in your program, you can be assured that the change has been saved to disk. The chance of lost data due to power interruptions, or disk crashes is much reduced. Also, since the entire database does not have to reside in memory at once, KirbyBase should run adequately on a memory-constrained system.
- You can specify the type of data that each field will hold. The available data types are: string, integer, float, boolean, date, and datetime.
- The query syntax is very "pythonic". Instead of having to use another language like SQL, you can express your query using a plain python expression using comparison operators (i.e. >50000). This works for all data types. Additionally, for strings, you can use regular expressions to express your query (i.e. '^Hawaii').
- All inserted records have an auto-incrementing primary key that is guaranteed to uniquely identify the record throughout it's lifetime.
- You can specify that the result set of a query be returned as a list of lists, dictionaries, or record objects.
- You can add and drop fields after you have created a table.

In meeting your DBMS needs, KirbyBase fits in somewhere between plain text files and small SQL database management systems like sqlite and Gadfly.

If you find a use for KirbyBase, I would love to get an email telling about how you use it. I find that hearing how people have put KirbyBase to use is the biggest reward I get for working on it.

Connecting to a database

To use Kirbybase, you first need to import the module:

```
from kirbybase import KirbyBase, KBError
```

Then create an instance:

```
db = KirbyBase()
```

By default, the instance is a local connection using the same memory space as your application. To specify a client/server connection, it would look like this:

```
db = KirbyBase('client', '192.168.0.10', '44444')
```

Of course, you would substitute the ip address and port number of your server and the port you are running the dbserver script on.

This method call will return a reference to a KirbyBase instance.

Creating a new table

To create a new table, you specify the physical file name, including path, that will hold the table, and a list containing the field names and field types. For example, to create a table containing information on World War II planes:

```
result = db.create('plane.tbl', ['name:str', 'country:str', 'speed:int',
 'range:int', 'began_service:datetime.date'])
```

Or, if the table needed to be in a different directory than the current directory, say in a directory called 'db' below the current directory (including the path in the table file name works for all KirbyBase methods):

```
result = db.create('./db/plane.tbl', ['name:str', 'country:str', 'speed:int',
 'range:int', 'began_service:datetime.date'])
```

Notice that in the fields list, you separate the field name from the field type with a colon. KirbyBase will automatically create a primary key field for each table called 'recno'. This field will be auto-incremented each time a record is inserted. You can use this field in select, update, and delete queries, but you can't modify this field.

Python field types currently allowed in KirbyBase are str, int, float, bool, datetime.date and datetime.datetime. To achieve compatibility between the Python and Ruby versions of KirbyBase, you can also use the following field types: String, Integer, Float, Boolean, Date, and DateTime. Use these instead of the above mentioned types if you plan on using your table from both Python and Ruby.

This method call will return True if the table was successfully created.

The insert method

To insert records into a table, you use the insert method:

```
recno = db.insert('plane.tbl', ['P-51', 'USA', 403, 1201, date.datetime(1943,05,27)])
```

The length of the data list must equal the length of the field list. Also, the data types must match. In the above example, specifying '403', instead of 403, would have resulted in an error.

You can also use a dictionary to specify field values. Rewriting the above example to use a dictionary instead of a list would look like this:

```
recno = db.insert('plane.tbl', {'name': 'P-51', 'country': 'USA',
'speed': 403, 'range': 1201, 'began_service': date.datetime(1943,05,27)})
```

You can also use an instance of the object class to specify field values. For each field of the record, you should have an attribute of the object instance, with it's value set to the value you want to insert into the table field. Rewriting the above example to use an object instead of a list would look like this:

```
class Record(object): pass

rec = Record()
rec.name = 'P-51'
rec.country = 'USA'
rec.speed = 403
rec.range = 1201
rec.began_service = datetime.date(1943,05,27)

recno = db.insert('plane.tbl', rec)
```

This method call will return the record number of the newly created record. This is an auto-incremented integer generated by KirbyBase. This number will never change for the record and can be used as a unique identifier for the record.

The insertBatch method

To insert multiple records into a table at one time, you use the insertBatch method:

```
recsToInsert = [
 ['P-51', 'USA', 403, 1201, date.datetime(1943,05,27)],
 ['P-47', 'USA', 379, 805, date.datetime(1942,01,11)]
]
recnoList = db.insertBatch('plane.tbl', recsToInsert)
```

This method call accepts a list of either lists or dictionaries, each one corresponding to a record you wish to insert into the table. It returns a list of unique record numbers, one for each record that KirbyBase inserted into the table.

How to select records

The syntax you use to select records to perform operations on is the same for a select, update, or delete statement, so I wanted to cover, in general, how to create a query expression first, before getting to the specifics of select, update, and delete.

For all data types **except string fields and boolean fields**, you can construct a query expression using simple Python comparison operators. **For string fields**, there are two ways you can build your query expression: (1) using implied exact matching and (2) using a regular expression.**For boolean fields**, you use implield exact matching (i.e. simply compare it to the boolean value True or False).

Here's a table showing the ways you can construct a query expression for each field type:

Type	Syntax allowed
int	==, =>, <=, >, <, <>, !=
float	==, =>, <=, >, <, <>, !=
date	==, =>, <=, >, <, <>, !=
datetime	==, =>, <=, >, <, <>, !=
str	exact match (i.e. 'John'), regular expression (i.e. '^John$')
bool	True, False

Selecting by int, float, date, and datetime field types

First, let's discuss using simple Python comparison operators. These include: ==, =>, <=, >, <, <> and !=. In each of the methods select, update, and delete, you simply place the comparison operator inside quotes, along with the value to compare it to (i.e. '>300'). Instead of having to hard-code the value to compare it to, you could use Python's string formatting operator to use variables in your code (i.e. '>%d'

% myvar). This expression syntax works for all fields, **except strings and booleans**.

Selecting by str field types

For string fields, you can select records by simply specifying a value that the field must match exactly. For example, if you want to select all records where the firstname field is "John", you would simply use the expression 'John'. This would select all Johns, but not Johnnys. **This is the default behavior for selecting by string fields. Note: This is a change in version 1.7. Prior to this version, the default behaviour was for string fields to be match using regular expressions, which is described next.**

There is an additional way to specify selection criteria for string fields: Python regular expressions. Any valid regular expression will work. So, for example, if you wanted to select all records where the fullname field begins with "John", you could express it as '^John'. This would return all records where the fullname field BEGAN with the letters "John", i.e. "John Jones", "Johnny Doe", but, because of the ^ symbol, would not include "Jack Johnson". To include him, you would need to change the query expression to 'John'. To enable regular expression matching in your queries, you need to pass the keyword argument useRegExp=True.

Ok, you might be asking yourself, "If it is correct to match a numeric or date field by doing '==45', why can't I do something like '==John' to match a string field. Why, for string matching do I have to do 'John' (or '^John$' if useRegExp=True)? Well, the answer is that for every field type except string, KirbyBase knows how to tell what is an operator in the expression versus what is a value. In other words, take the expression '==45'. If this is being matched against an integer field, KirbyBase knows that the '==' is the operator meaning equals and it knows that the rest, 45 is the value to check against. But what if the expression '==45' is being matched against a string field? If I allow the user to use comparison operators in expressions matching against string fields, then I don't know if the user is looking for a string field value that is equal to '45' or '==45'. That's why I don't allow comparison operators in string field match expressions. To be honest, I haven't put a ton of thought into it. I'm sure, if it became an issue, I could find a (hopefully) elegant solution.

Selecting by bool field types

For boolean fields, you can specify selection criteria by simply comparing the field to the built-in names True or False.

Multiple selection criteria

You can have multiple selection criteria in one query expression. For example, to select all records with "John" as the firstname and with salary greater than $50,000, you could specify both 'John' and '>50000' in your select statement. This will be shown more clearly in the select examples below.

Selecting multiple records by record number

You can select multiple records by including a list of record numbers to select. You will see an example of this in the select examples below.

Selecting all records

Finally, there is one specialized way to select all records from a table. In the query expression, if you select against the recno field and give the selection criteria as '*', KirbyBase will return all table records.

Now that we have a general understanding of how to select records to operate on, lets get more specific by looking at the select, update, and delete methods.

The select method

The select method allows you to ask for all records in a table that match certain selection criteria. Additionally, you can also specify which fields you want included in the result set, whether you want the result set sorted, and the format of the records in the result set (i.e. list, dictionary, or object).

A simple example of a select statement would be:

```
result = db.select('plane.tbl', ['country'], ['USA'])
```

This statement is asking for every record in the plane table that has a country field equal to 'USA'. Since the default is to not use regular expression matching, it will look for an exact match to 'USA'. Because we did not specify which fields were to be included in the result set, all the fields of a record will be returned. Also, since we did not specify a sort order, the result set will be unsorted.

Using Regular Expressions

Now, let's say some of the values in the country field have 'USA' and some have 'United States'. How can we tell KirbyBase we want to match both values? One way would be to explicitly specify both in our statement:

```
result = db.select('plane.tbl', ['country','country'], '['USA','United States'])
```

Another way would be to use a regular expression:

```
result = db.select('plane.tbl', ['country'], ['US|United States'], useRegExp=True)
```

Notice how we had to supply the keyword argument **useRegExp** and set it to **True**, in order to have KirbyBase match based on regular expressions.

Using Comparison Operators

Now lets take a look at some selects using comparison operators. To select all planes that have a maximum speed between 300 mph and 400 mph:

```
low_speed = 300
high_speed = 400
result = db.select('plane.tbl', ['speed', 'speed'], ['>%d' % low_speed,
 '<%d' % high_speed])
```

Notice that to select a numeric range, all I had to do was use the same field twice in the statement and give it greater than and less than

values. Also, notice that you can easily use Python variables instead of hard-coded values by taking advantage of Python's string formatting operator: %.

Selecting based on multiple fields

How about all planes that are fast AND have a long range?

```
result = db.select('plane.tbl', ['speed', 'range'], ['>350', '>900'])
```
You can, of course, combine both string and non-string comparisons in the same select statement. This would allow you to select, for example, all bombers with a speed greater than 300mph:

```
result = db.select('plane.tbl', ['plane_type', 'speed'], ['^Bomber', '>300'],
 useRegExp=True)
```
Notice how we used the ^ symbol to tell KirbyBase that we want only those records where the plane_type field BEGINS with the word Bomber. This ensures that we don't get 'Fighter-Bomber' records included in our result set. The ^ is regular expression syntax that means "match from start of string". All regular expression syntax will work with KirbyBase.

Selecting by date or datetime fields

Let's select records using the date field type. To select all planes that entered service before 1940:

```
result = db.select('plane.tbl', ['began_service'], ['<%s' % datetime.date(1940, 1, 1)])
```
Notice that you use %s with date and datetime field comparisons. This is because, inside KirbyBase, date/datetime comparisons are done by coercing them to strings instead of actually comparing date/datetime objects. The reason this is done is one of speed (if you want the gory details, I have quite a long comment in the source code that explains the reasoning behind this). The end result is the same because when doing comparisons against two date/datetime fields that have been coerced into strings gives you the same answer as comparing the two fields in their original date/datetime format.

Selecting by boolean fields

Now, lets see how to select records using booleans. Here's how we would select all planes that are still flying:

```
result = db.select('plane.tbl', ['still_flying'], [True])
```
Selecting by record number(s)

To select an individual record using it's record number:

```
result = db.select('plane.tbl', ['recno'], [245])
```
To select multiple records by record number:

```
result = db.select('plane.tbl', ['recno'], [2,5,7])
```
Selecting all records in the table

Finally, you can select all records easily by doing the following:

```
result = db.select('plane.tbl', ['recno'], ['*'])
```
Specifying the type of result set

The select method will return a list of records. Each result record can either be a list, a dictionary, an object, or a line formatted for pretty printing. The default record type is list. For example, to select all planes, but have each returned record be a dictionary:

```
result = db.select('plane.tbl', ['recno'], ['*'], returnType='dict')
for record in result:
    print record['name']
```
To select all planes and have each returned record be an object:

```
result = db.select('plane.tbl', ['recno'], ['*'], returnType='object')
for record in result:
    print record.name
```
If you specify 'report' as the returnType, the result of a select will be returned in a nice tabular format, suitable for printing. For example, the following code:

```
print db.select('plane.tbl', ['recno'], ['*'], ['recno','name','country',
 'role'], returnType='report')
```
will result in the following being returned from select:

```
recno | name      | country | role
-------------------------------------------
    1 | P-51      | USA     | Fighter
    2 | P-47      | USA     | Fighter
    3 | B-17      | USA     | Bomber
    4 | Typhoon   | England | Fighter-Bomber
    5 | Sptitfire | England | Fighter
    6 | Oscar     | Japan   | Fighter
    7 | ME-109    | Germany | Fighter
    8 | JU-88     | Germany | Bomber
   10 | Zero      | Japan   | Fighter
```
You can control how many records are printed before a formfeed (\f) is emitted and whether dashed lines are printed between records by supplying a a two-element list as the keyword argument rptSettings. The first item in the list must be an integer specifying how many records to print before inserting a formfeed character. The default is 0, which means don't insert any formfeeds at all. The second item of the list is a boolean. If true, a line of dashes is inserted between each record. The default is false.

Limiting the fields in the result set

By default, the result set records will have all of the fields of the table record. If you only want certain fields for each record returned, you can specify a filter list. For example, to select all planes from the USA, but you only want their name and speed:

```
result = db.select('plane.tbl', ['country'], ['USA'], ['name', 'speed'])
```

Sorting the result set

You can also specify that you want the result set to be sorted by one or more fields and, for each sort field, you can specify the sort direction. The sort fields argument must be a list containing valid field names. The result set will be sorted first by the initial field name in the list, then subsequently by each additional field. So, specifying a sort field list of ['country','role','name'] will sort the result set so each country's planes will be grouped together with Germany's records coming before the USA's records, and within each country group, bombers will be grouped together before fighters and within bombers the records will be sorted by name. If you pass a filter list to the select method, the field names specified in the sort fields list must be members of the filter list. In other words, you can't sort on a field that is not included in the result set. So, for example, to sort the result set by name within country would look like this:

```
result = db.select('plane.tbl', ['recno'], ['*'], sortFields=['country', 'name'])
```

If a sort field(s) is specified, the sort direction will default to ascending; to sort in a descending direction you would pass a list as the sortDesc argument. In that list would be the field names you want to be sorted in descending order. Each item in sortDesc must be a member of the list you passed as sortFields. In other words, you can't specify a field to sort in descending order if you have not specified that you want to sort by that field in the first place. So, if you wanted to sort the result set by role and speed, but have the fastest planes for each role appear first, you would code the following:

```
result = db.select('plane.tbl', ['recno'], ['*'], sortField=['role',
 'speed'], sortDesc=['speed'])
```

The update method

To update a table, you must specify the criteria for determining which records will be updated. To update a single record, you can specify it's recno. This will search the table ONLY by this field and will update, at most, one record. An example of this would be:

```
result = db.update('plane.tbl', ['recno'], [54], [405], ['speed'])
```

Here we are updating the record with recno equal to 54. We are changing it's speed to 405 mph. Specifying an exact recno to match is generally a much faster transaction, because KirbyBase will match only on that field and it will return as soon as that record has been found, instead of searching through the entire table.

You can also specify match criteria for one or more fields using either regular expressions (for string field types) or comparison expressions (for int,float field types). You can mix and match. For example, to update all USA planes with a speed greater than 400 mph:

```
result = db.update('plane.tbl', ['country', 'speed'], ['USA', '>400'],
 [1500], ['range'])
```

This example will change the range to 1,500 miles for all USA planes with a speed greater than 400mph.

Updating all fields in a record at one time

The update criteria is specified as a list of data values to set the fields to, called updates, and a list of which fields to update, called a filter. You can, of course, update multiple fields at one time. If the filter list is equal to None, then the updates list MUST have a value for each field in the record, excepting the 'recno' field (which can never be updated). An example of this would be:

```
result = db.update('plane.tbl', ['recno'], [106], ['P-47', 'USA', 347, 789,
 datetime.date(1942,12,22)])
```

By specifying an update list that is equal in length and in the same order as the record's fields, you then do not have to specify which fields you are going to update. KirbyBase will simply appy each item in the update list to it's corresponding field in the record.

Updating using a dictionary

You can also use a dictionary to specify the updates to field values. Rewriting the above example to use a dictionary instead of list would look like this:

```
result = db.update('plane.tbl', ['recno'], [106], {'name': 'P-47', 'country': 'USA',
 'speed': 347, 'range': 789, 'began_service': datetime.date(1942,12,22)})
```

Updating using a record object

You can also use an instance of the object class to specify the updates to field values. For each field you want to update, you should have an attribute of the object instance, with it's value set to the value you want to update the table field. Rewriting the above example to use an object instead of list would look like this:

```
class Record(object): pass

rec = Record()
rec.name = 'P-47'
rec.country = 'USA'
rec.speed = 347
rec.range = 789
rec.began_service = datetime.date(1942,12,22)

result = db.update('plane.tbl', ['recno'], [106], rec)
```

The update method call will return an integer specifying the total number of records that were updated.

The delete method

Deleting records from a table is similar to updating, except that you don't specify an update list or a filter list.

As mentioned above, to delete a specific record, just specify 'recno' as the field to search on and put the exact record number in the patterns list. So, to delete the record with recno 456:

```
result = db.delete('plane.tbl', ['recno'], [456])
```
To delete multiple records, you would specify match criteria. This works exactly the same as it does for updates and selects. Therefore, to delete all planes from Germany that have a range less than 800 miles:

```
result = db.delete('plane.tbl', ['country', 'range'], ['Germany', '<800'])
```
This method call will return an integer specifying the total number of records that were deleted.

The pack method

When KirbyBase deletes a record, it really just fills the entire line in the file with spaces. Deleting the entire line and moving each subsequent line up one would take too much time. Also, when a record is updated, if the size of the updated record is greater than the size of the old record, KirbyBase spaces out that entire line in the file, and rewrites the updated record at the end of the file. Again, this is done so that the entire file doesn't have to be re-written just because one record got updated.

However, this means that after a lot of deletes and updates, a table can have lots of blank lines in it. This slows down searches and makes the file bigger than it has to be. You can use the pack method to remove these blank lines:

```
result = db.pack('plane.tbl')
```
This method call will return an integer specifiying the number of blank lines that were removed from the table.

The validate method

KirbyBase will validate data added through the insert or update methods to ensure that it is of the proper data types, but what if you add data to a table by other means? For example, one of the nice things about KirbyBase is that you can simply open it in a text editor and add or change data manually.

Obviously, we need a way to validate that the data in a table is still of the correct data types. This is the purpose of the validate method. The method is invoked like so:

```
result = db.validate('plane.tbl')
```
The result set will contain a list of records that had invalid data. Each record in the list will contain the record number, the field name, and the invalid data for that field. An empty list means that the table passed the validation test.

The drop method

To delete a table, including the physical file that holds it, use the drop method:

```
result = db.drop('plane.tbl')
```
This method call will return True on success.

The addFields method

To add a new field(s) to a table, use this method:

```
db.addFields('plane.tbl', ['bomb_load:int'], after='range')
```
The above example adds a new field called bomb_load with a type of int to the Plane table. The 'after' keyword allows you to tell KirbyBase where to insert the new field. If you don't specify a field name for after, the new field will be inserted right after the built-in recno field.

This method call will return True on success.

The dropFields method

To remove a field(s) to a table, use this method:

```
db.dropFields('plane.tbl', ['bomb_load:int'])
```
The above example removes the field called bomb_load from the Plane table.

This method call will return True on success.

Miscellaneous methods

```
result = db.getFieldNames('plane.tbl')
```

Result will be a list of the table's field names.

```
result = db.getFieldTypes('plane.tbl')
```
Result will be a list of the table's field types(i.e. 'str', 'int', 'float')

```
totalRecords = db.len('./db/plane.tbl')
```
Result will be an integer showing the total number of records in the table.

```
db.setDefaultReturnType('object')
```
The default return type for select queries is 'list'. You can override this on an individual select basis by using the returnType keyword in the select method, but this can get tiring if you want to always have a different return type. This method allows you to set the default return type for the select method to something els.

Special characters in data

Since KirbyBase tables are just plain-text, newline-delimited files with each field delimited by a '|', certain ASCII characters could cause problems when used as input. For example, entering a newline character (\n on Unix, \r\n on Windows) as part of a record's data would cause problems later when KirbyBase attempted to read this record. Likewise, using the '|' character in your data would also cause problems as KirbyBase uses this character as a field delimiter. Finally, it turns out that Python has problems handling octal code \032 under Windows (possibly because it equates to Ctrl-Z).

To handle the above special characters as data input, KirbyBase checks all string input data and replaces the special characters with encodings that are safe. The following table shows how replacements are done:

Input Character	KirbyBase Replacement
\n	&linefeed;
\r	&carriage_return;
\032	&substitute;
|	&pipe;

KirbyBase will translate to and from the special characters as data is written to and read from a table. It should all be transparent to the user. The only time you would encounter the replacement words is if you were to open up the physical table file in a text editor or read it in as input outside of KirbyBase.

Table Structure

Every table in KirbyBase is represented by a physical, newline-delimited text-file. Here is an example:

```
000006|000000|recno:int|name:str|country:str|speed:int|range:int
1|P-51|USA|403|1201
2|P-51|USA|365|888
3|Sptitfire|England|345|540
4|Oscar|Japan|361|777
5|ME-109|Germany|366|514
6|Zero|Japan|377|912
```
The first line is the header rec. Each field is delimited by a '|'. The first field in the header is the record counter. It is incremented by KirbyBase to create new record numbers when records are inserted. The second field in the header is the deleted-records counter. Every time a line in the file is blanked-out (see Pack), this number is incremented. You can use this field in a maintenance script so that the table is packed whenever the deleted-records counter reaches, say, 5,000 records. The third field in the header is the recno field. This field is automatically added to the table when it is created. The rest of the fields are whatever you specified when you created the table.

Each record after the header record is simply a line of text. Newline characters delimit records.

Server Notes

There are two server scripts included in the distribution:

kbsimpleserver.py - creates a single-threaded, blocking server. This means that client requests are serviced in sequential order. If one client does a 10,000 record select, the client request queued behind it for updating one record will have to wait, even if it is going against a different table. This server is good with small tables and clients that do small, simple queries.

kbthreadedserver.py - creates a multi-threaded, non-blocking server. This means that each client that connects runs in it's own thread on the server. The only time one thread will block the others is if it has to open a table in write mode (i.e. db.delete, db.update, db.insert). Even in this case, it will only block other write requests. Any selects will not be blocked because they do not need exclusive access to the table. And, unlike the simple server script, this server only blocks if two clients are trying to write to the SAME table. If one client is updating the plane.tbl and another client is trying to update the tank.tbl, they will not block each other. The script does this by maintaining a list of table locks, instead of just one lock for the whole script.

***Notice: Although I have tested kbthreadedserver.py some, I would still label it beta. Try it out with some test data and see how it works for you before trying it in a production environment.

License

KirbyBase is licensed under the Python Software Foundation License.