GNUe Forms: A Developer's Introduction

A Guide to Programming with GNUe Forms

Version 0.5.1

Written by Jason Cater, your tour guide

Copyright © 2000-2003 Free Software Foundation.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled *GNU Free Documentation License*.

Table of Contents

Introduction **4**
- Structuring the Database 4
- Designing the Form 4
- Planning for Security 4

Basic Concepts **5**
- Datasources 5
- Blocks and Fields 5
- Pages and Visual Elements 5
- Triggers 5
- Designing for Multiple Interfaces 5
- Designing for Multiple Databases 5

Creating your First Form **6**
- Preliminary Steps 6
- Creating the Empty Form 7
- Creating the Datasources 7
- Creating the Logic Structure 8
- Creating the Layout 9
- Running the Form 11
- Where To Go Next 11

Understanding Datasources **12**
- Table-Bound Datasources 12
- Static Datasources 12
- Defining Conditions 13
- Linking Datasources via Master/Detail 13
- Advanced Relationships 13

Understanding Events and Triggers **14**
- Form's Event Model 14
- Named Triggers verses Embedded Triggers 14
- Form-level Triggers 14
- Block-level Triggers 15
- Field-level Triggers 18
- Page-level Triggers 19
- Entry-level Triggers 19
- Button-Level Triggers 19

Working with Fields **21**
- Typecasting Fields 21
- Default Values 21
- Formatting Fields with Masks 21
- Dropdown Fields 24
- Check boxes 24

A Brief Introduction to Python 25
The Basics 25
Variables and Expressions 25
Control Structures 25
Tuples, Lists, and Dictionaries... oh, my! 26

Exploring Trigger Namespaces 27
Introduction 27
Object Heirarchy 28
Fields and Entries 28
Blocks 28
Datasources 28
Form 29

Creating and Using Libraries 30
Overview 30

Integration with GNUe Tools 31
Running Reports from Forms 31
Running Forms from Navigator 31

Advanced Topics 32
Runtime Parameters 32
External Python Modules 33
Designing for Multiple Interfaces 33

Trigger Recipes 35
Timestamping a Record prior to a Commit 35
Stamping a Record with User's Login prior to a Commit 35
Auto-Populating an Entry from a Sequence 36

Appendix A: Trigger Hierarchy 37

Appendix B: Form Elements 38
Form Tags 38
Connection Tags 38
Datasource Tags 39
Dialog Tags 43
Layout Tags 43
Logic Tags 46
Menu Tags 48
Options Tags 49
Parameter Tags 50
Trigger Tags 50
Import Tags 51

Appendix C: Form Objects 56

Form....56
Datasource....59
Block....60
Entry....62

Appendix D: Data Objects....65
Result Set....65
Record Set....65

Appendix E: Sample Librarian Schema....66

Appendix F: Glossary....67

Appendix G: GNU Free Documentation License....68

Introduction

This section briefly introduces the process of designing an application using GNUe Forms. BLAH, BLAH, BLAH...

Before designing an application for Forms, the developer should be somewhat familiar with a few key concepts:

- *Database Design* - This guide does not delve into database design. It is assumed the developer can either create his own tables, or has an existing set of tables to work with.
- *Python Scripting* - GNUe uses Python for scripting/event support. Any level of serious applications programming will likely require some level of Python. There is a short section entitled "A Brief Introduction to Python" in this guide that can serve as a starting point. The average programmer can likely learn enough simply by trying out the examples in this guide.
- *XML* - GNUe extensively uses XML for its internal storage format. While it is possible to create GNUe applications via Designer without interacting with the XML formats, a good, solid understanding of XML basics would definitely be useful.

Structuring the Database

TODO

Designing the Form

TODO

Planning for Security

TODO

Basic Concepts

TODO

Datasources

Since GNUe Forms was designed from the ground up to manipulate database data, it must have some way to tie the graphical elements to database tables. This is where a `datasource` comes into play.

A `datasource` provides a link to a database table or some similar data store.

[TODO: EXPAND]

`Datasources` are portable from one GNUe tool to another. A `datasource` that is usable in forms should also be usable in a report.

Blocks and Fields

A datasource provides a link to a table, but Forms needs more information than a simple reference to meaningfully interface with an end user. A `block` is the first step to making `datasource`s suitable for such user interaction. At its most basic level, a `block` contains instructions on how Forms should interact with a `datasource`.

Any `datasource`s that are to interact with a user must have a single corresponding `block`. `Datasource`s that are only used internally and not for displaying data will not normally have an associated `block`, although the developer may choose to do so.

A field is ....

Pages and Visual Elements

TODO

Layout Management

TODO

Triggers

TODO

Designing for Multiple Interfaces

TODO

Designing for Multiple Databases

TODO

Creating your First Form

In this chapter, we will create our first full-featured form -- a postal code lookup table.
[TODO: Expand]

[TODO: Replace the postal code example with librarian]

We are going to create a form that looks something like:

```
01234567890123456789012345678901234567890
|---------+---------+---------+---------|

  Zip Code:   [___]

  City Name:  [_________________________]

  State:      [__]
```

The ruler is simply for our layout reference. We will come back to it later.

[TODO: expand]

Preliminary Steps

[TODO: Intro]

The following two steps may need to be performed by your database or systems administrator.

First, we need to create a test table. Using your database of choice, create a table named `zipcodes`, with three appropriately sized fields, `zipcode`, `city`, and `state`. In PostgreSQL, SAP-DB, and others, the following statement will work:

```
create table zipcodes (
  zipcode varchar(5),
  city varchar(30),
  state varchar(2) );
```

For other databases, create a similarly structured table.

Next, if you have not done so yet, setup your connections.conf file to point to your database. Our examples will use a connection called `tutorial`. An example entry:

```
[tutorial]
comment = Tutorial Database
provider = psycopg
host = localhost
dbname = mydb
```

Of course, your entry will probably look different. This example is using the PostgreSQL `psycopg` driver, connecting to a database named `mydb` running on the local machine. See the *Forms Installation Guide* for information on the location and syntax of this file.

If you already have a connection for the database you will be using, simply add an `alias = tutorial` line to the appropriate section. Example:

```
[dev]
comment = Foobar Development
provider = psycopg
host = dbserver
dbname = mydb
alias = tutorial
```

Creating the Empty Form

By default, GNUe Designer will start up with an empty form. You may also create a new form by selecting `File | New | Form`.

[TODO: CHANGE THE TITLE OF THE FORM]

Creating the Datasource

Our sample form will use a single table - `zipcodes`. To access this table in Forms, we need to associate it with a datasource. We will call this datasource `MyDS`.

From the menu, select `Edit | Insert | Datasource...` to run the datasource wizard. The first step of the datasource wizard asks for your connection. Select tutorial. This is the connection we setup in the first part of this section. Click `Next...` You may be asked to log in. Provide a valid database username and password.

It will need three attributes: `name`, `connection`, and `table`.

- `name`: All referencable objects in GNUe need a unique name. We will name our datasource `MyDS`.
- `connection`: This is the name of the connection we set up in the first part of this section. If you used our example entry, then this is called `tutorial`.
- `table`: This should be the name of the table we will be referencing. Once again, if you used our example table create script, this will be `zipcodes`.

Creating the Logic Structure

[TODO: EXPLAIN THE LOGIC SECTION AND TAG]

Blocks are the display equivalent of datasources. Since we are working with a single datasource, we will have a corresponding single block. Since this is tied to the `zipcodes` table, we will call this block `ZipBlock`.

Block-specific attributes:

- `name`: Set this to a unique name that we will later refer back to.
- `datasource`: The name of the datasource that this block is tied back to. If no `datasource` is specified, then the block will be considered an *unbound* block.

Field-specific attributes:

- `name`: Set this to a unique name that we will later refer back to.
- `field`: Set this to the name of the corresponding field in the database.
- `case`: This can be set to any one of `mixed`, `upper`, or `lower` to force case convention. For example, if set to `upper`, then all lower case input characters will be converted to uppercase. Note that this only applies to data input from the user. Setting this field will not convert existing data that is queried. The default value is `mixed`.

.....

Creating the Layout

Forms contains its layout logic in units called *pages*. Only a single page is normally seen any given time by the end user. Our simple form will need one page. We will call it `MyPage`.

Given the layout grid we created earlier, we see that our form will be 40 characters wide and 7 lines high. For simplicity's sake, we are using a simple character based layout, identified as `GNUe:Layout:Char`. In the future, forms will support other layout styles.

Again, looking back at our earlier layout grid, we have three labels and three entries. Each label starts in column two and each entry starts in column 13. Each pair of label/entry skips a row, with the first pair being on row 1.

This gives us enough information to create our display layout.

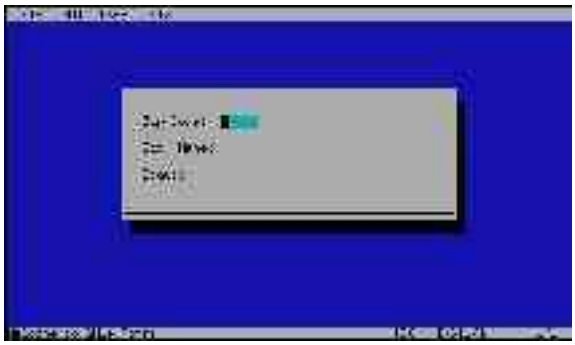
Via GNUe Designer:

An empty form created by GNUe Designer automatically has a single, empty page. If we were to add more pages, this could be accomplished by selecting `Edit | Insert | Page`. However, this example uses a single page, so we do not need to do anything for this step.

[TODO]

Running the Form

TODO



[TODO: Insert pic of Win32 form]

[TODO: Insert pic of HTML form]

Where To Go Next

[TODO: Well, home *is* where the heart's at]

Understanding Datasources

A Datasource links data to our form. Usually, a datasource points to a table if using a relational database, or a data object if using an object database. A form can have several datasources if pulling data from multiple locations, or no datasources at all if the form does not reference outside data.

If a form does not have a datasource, a *virtual datasource* is created. The *commit*, *rollback*, and *query* functions do not serve a purpose against virtual datasources. This is particularly useful for *action forms* that simply cause actions to occur, but do not directly manipulate data.

Datasources can be linked to each other in a *master/detail* fashion via a foreign key. In essence, each time the *master* datasource changes, the *detail* datasource is automatically requeried to bring up records related to the *master*. See the section on master/detail relationships for more information.

Table-Bound Datasources

The most common datasource is one that is bound to an individual table or view.

Static Datasources

TODO

```
<datasource name="AvailDS" type="static">
  <staticset fields="id,descr">
    <staticsetrow>
      <staticsetfield name="id" value="A"/>
      <staticsetfield name="descr"
                      value="Available"/>
    </staticsetrow>
    <staticsetrow>
      <staticsetfield name="id" value="N"/>
      <staticsetfield name="descr"
                      value="Not Available"/>
    </staticsetrow>
    <staticsetrow>
      <staticsetfield name="id" value="B"/>
      <staticsetfield name="descr"
                      value="Backordered"/>
    </staticsetrow>
  </staticset>
</datasource>
```

TODO

Defining Conditions

Form's datasources support conditions. Conditions place restrictions on the records returned by a datasource. For those familiar with SQL, a condition translates directly into a `WHERE` clause.

```
<datasource connection="test" table="reps">
  <condition>
    <or>
      <eq>
        <cfield name="active"/>
        <cconst value="Y"/>
      </eq>
      <gt>
        <cfield name="sales_ytd"/>
        <cconst value="0"/>
      </gt>
    </or>
  </condition>
</datasource>
```

In this example, we are basically only allowing records from the `reps` table where the representative is either active (`active = 'Y'`) or had sales this year.

Linking Datasources via Master/Detail

Quite often, you will want a second datasource's behavior to be tied to a primary datasource. If a new record is queried in the first datasource, all corresponding records in the second datasource should automatically appear. Likewise, creating a new record in the first datasource should clear out the second datasource and any subsequent new records in this second datasource will be automatically associated with the new primary record.

In GNUe, we call this relationship a Master/Detail relationship. It closely mirrors the concept of primary/foreign keys in relational databases.

Master/detail relationships have the following properties:

- When a new record is created in the master datasource, the detail datasource will have no records.
- When the master source is queried, a new set of records is queried in the child datasource for each record in the master source. This happens on an as-needed basis so as not to waste resources.
- A master record cannot be deleted while detail records exist.
- On posting, or saving, a detail record, its “foreign key” is automatically populated with the “primary key” of the master record.

Defining Master/Detail Datasources

There are no special attributes for a master datasource to indicate its role as master.

The detail datasource, on the other hand, has three special attributes that must be provided: `master`, `masterlink`, and `detaillink`.

- `master`: This should be the name of the master datasource.
- `masterlink`: This is the name of the field or fields in the master datasource that links it to the child datasource. If there are multiple fields, i.e., the master primary key is "composite", then `masterlink` should be a comma-separated list of field names.
- `detaillink`: This is the name of the field or fields in the detail datasource that links it to the child datasource. If there are multiple fields, i.e., the master primary key is "composite", then `detaillink` should be a comma-separated list of field names with the order of the fields corresponding to the order provided in `masterlink`.

Master/Detail Considerations

TODO

Advanced Relationships

TODO

Master/Detail/Detail

TODO

Reverse Master/Detail

TODO

Understanding Events and Triggers

TODO

Form's Event Model

TODO

Named Triggers verses Embedded Triggers

TODO

Form-level Triggers

A Form-level trigger is defined as an object that is activated at the form-level and is defined as a child of the form object.

The following form level triggers are defined.

On-Startup

The `On-Startup` trigger is executed once during the lifetime of a Form's instance. This happens after all the objects have been initialized and initially populated.

Suggested uses for `On-Startup`:

- Setting initial flags.
- Marking variables or common functions/modules as:

On-Activate

The On-Activate trigger is executed each time a form or dialog is activated. For a normal form, the difference between `On-Activate` and `On-Startup` is very nominal. However, for `dialog`-style forms, the difference is more pronounced. `On-Startup` will only be executed once in the form's life, whereas `On-Activate` is called every time the dialog is instantiated and displayed.

For non-`dialog` forms, we recommend you use `On-Startup`.

Suggested used for `On-Activate`:

- Useful for adjusting dialogs based upon parameters passed in by the calling form.

On-Exit

The `On-Exit` trigger is executed when either the user or a trigger requests that a form closes.

Suggested uses for `On-Exit`:

- Wishing the user best of luck?

Pre-Commit

The `Pre-Commit` trigger is executed before a form-level commit occurs.

Suggested uses for `Pre-Commit`:

- Perform form-level validation of data (??)
- [TODO: ???]

Post-Commit

The `Post-Commit` trigger is executed after a form-level commit occurs.

Suggested uses for `Post-Commit`:

- [TODO: ???]

Block-level Triggers

All block-level triggers are executed on a per-record basis. That is, a trigger would get executed once for every applicable record, not just once for the entire block.

Pre-Query

The `Pre-Query` trigger is executed before a query against the database occurs. This trigger is unique from all other triggers in that it is called while the form is in *query* mode -- ie., the same mode as selecting `Data` | `Enter Query` from the menu. This means that any field changes made by this trigger don't actually modify a record, but instead are used as query conditionals.

Suggested uses for a block-level `Pre-Query` trigger:

- Adding custom conditions to a query that are more complex than can be represented by a `field`'s `queryDefault` property.

Post-Query

The `Post-Query` trigger is executed after a query occurs on a record. A block-level `Post-Query` is executed once for each record that was loaded from a database. `Post-Query` is the counter-part to `On-NewRecord` in that one or the other should be executed for every displayed record.

NOTE: `Post-Query` is only fired as a record is loaded from the database. This implies that, with GNUe's caching system, if only 10 records are displayed on screen at a time out of a table of 100 records, then only the first 10 or so records will have `Post-Query` fired. It is guaranteed, however, that by the time a the user can see a loaded record or before another trigger can be fired against a loaded record, `Post-Query` has already been called. It is possible to get around this by, at some point [TODO: WHERE?], calling `block.lastRecord()` and (optionally, if needed) `block.firstRecord()`.

Suggested uses for a block-level `Post-Query` trigger:

- Populating non-database (automaticall calculated) fields.
- Resetting user-defined flags

Pre-Change

The `Pre-Change` trigger is executed before a record is initially modified -- i.e., when the first field of a record is set to a new value. At the time the Pre-Change record is called, the modified field will still contain the old value.

Suggested uses for a block-level `Pre-Change` trigger:

- [TODO: A GOOD EXAMPLE?]

Pre-Insert

The `Pre-Insert` trigger is executed before a commit occurs on a record that is pending an insertion. A block-level `Pre-Insert` is executed once for each inserted record and is fired prior to the `Pre-Commit` trigger.

Suggested uses for a block-level `Pre-Insert` trigger:

- Stamping records with a creation date or created-by value
- Setting a primary key's value
- Setting other hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-Update

The `Pre-Update` trigger is executed before a commit occurs on a record that is pending an update. A new or deleted record is not considered "updated" for the purpose of this trigger. A block-level `Pre-Commit` is executed once for each changed record and is fired prior to the `Pre-Commit` trigger.

Suggested uses for a block-level `Pre-Update` trigger:

- Stamping records with a modification date or modified-by value
- Setting hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Pre-Delete

The `Pre-Delete` trigger is executed before a commit occurs on a record that is pending a deletion. A block-level `Pre-Commit` is executed once for each record that has pending deletion. A block-level `Pre-Delete` fires prior to a `Pre-Commit` trigger.

Suggested uses for a block-level `Pre-Delete` trigger:

- [TODO: A GOOD EXAMPLE?]
- Storing historical information in transaction tables

Pre-Commit

The `Pre-Commit` trigger is executed before a commit occurs on a record. A block-level `Pre-Commit` is executed once for each record that has pending changes, including new and deleted records. A block-level `Pre-Commit` fires prior to a Field's `Pre-Commit` trigger, but after the `Pre-Insert`, `Pre-Update`, and `Pre-Delete` triggers.

Suggested uses for a block-level `Pre-Commit` trigger:

- Stamping modified records with a date or modified-by value
- Setting hidden, but pertinent, fields with default or pre-calculated values
- Storing historical information in transaction tables

Post-Commit

The `Post-Commit` trigger is executed after a commit occurs on a record. A block-level `Post-Commit` is executed once for each record that had pending changes, including new and deleted records. A block-level `Post-Commit` fires after a Field's `Post-Commit` trigger.

If a `Post-Commit` trigger modifies the values in a record, then the record will be, once again, pending changes. Typically a `Post-Commit` trigger would not modify any values and you could create an unsavable form.

Suggested uses for a block-level `Post-Commit` trigger:

- Resetting user-defined flags or non-database fields.

On-NewRecord

The `On-NewRecord` trigger is executed when a record is initially created. This trigger is executed once for each new record at the time of creation in the form.

Suggested uses for a block-level `On-NewRecord` trigger:

- Setting default values

Pre-FocusIn

The `Pre-FocusIn` trigger is executed as a new record is focused in a block. It is recommended that unless you have a specified understanding of the intention of forms, use `Post-FocusIn` instead of `Pre-FocusIn` as the latter trigger's behavior may change at some point to better reflect record focus.

Suggested uses for a block-level `Pre-FocusIn` trigger:

- [TODO: ???]

Post-FocusIn

The `Post-FocusIn` trigger is executed as a new record is focused in a block. This may be triggered by a user navigating to a different record or by creating a new record. The actual record change has occurred when this trigger is fired.

Suggested uses for a block-level `Post-FocusIn` trigger:

- [TODO: ???]

Pre-FocusOut

The `Pre-FocusOut` trigger is executed as a different record is about to be focused in a block. This may be triggered by a user navigating to a different record or by creating a new record. The actual record change has not occurred when this trigger is fired.

Suggested uses for a block-level `Pre-FocusOut` trigger:

- [TODO: ???]

Post-FocusOut

The `Post-FocusOut` trigger is executed as a new record is focused in a block. It is recommended that unless you have a specified understanding of the internals of forms, use `Pre-FocusOut` instead of `Post-FocusOut` as the latter trigger's behavior may change at some point to better reflect record focus.

Suggested uses for a block-level `Post-FocusOut` trigger:

- [TODO: ???]

Field-level Triggers

TODO

Pre-FocusIn

TODO

Post-FocusIn

TODO

Pre-FocusOut

TODOglimpse

Post-FocusOut

TODO

Post-Query

TODO

Pre-Modify

TODO

Pre-Insert

TODO

Pre-Update

TODO

Pre-Delete

TODO

Pre-Commit

TODO

Post-Commit

TODO

Pre-Change

TODO

Post-Change

TODO

Page-level Triggers

TODO

Pre-FocusIn

TODO

Post-FocusIn

TODO

Pre-FocusOut

TODO

Post-FocusOut

TODO

Entry-level Triggers

TODO

Pre-FocusIn

TODO

Post-FocusIn

TODO

Pre-FocusOut

TODO

Post-FocusOut

TODO

Button-Level Triggers

Buttons have a special relationship with triggers.

On-Action

This trigger is run when the user activates (e.g., clicks) a button.

Suggested uses for an `On-Action` trigger:

- Perform a calculation
- Navigate to another section of the form
- Open another form

Pre-FocusIn

TODO

Post-FocusIn

TODO

Pre-FocusOut

TODO

Post-FocusOut

TODO

Post-Change

TODO

Working with Fields

TODO

Typecasting Fields

TODO

Default Values

TODO

Formatting Fields with Masks

[NOTE: Format Masks are not yet completely functional! This section reflects the intended support]

Forms supports two types of format masks: display masks and input masks. A display mask defines how the field data will be formatted for display. An input mask defines how the user will edit a field's value. Input mask elements are a subset of display mask elements -- in other words, all input masks can also be used as display masks, but not all display masks can be used as input masks.

Note: if first character of a format is '&', then rest of date defines a preset format (settable by developer? in gnue.conf or geas?).

e.g., in gnue.conf:

```
FormatDate_longdate = "A, b d, Y"
```

Then, in the client, the format string could be: `&longdate`

This allows reuse of common format masks throughout the application.

Formatting Numeric Fields

TODO

Formatting Date/Time Fields

Element	Input?	Description
\	Yes	Next character is a literal
a	Yes	Abbreviated weekday name (Sun..Sat)

A		Full weekday name (Sunday..Saturday)
b	Yes	Abbreviated month name (Jan..Dec)
B		Full month name (January..December)
c		Century (20,21)
d	Yes	Day of month, left padded with zeros (01..31)
D		Day of month, non-padded (1..31)
h	Yes	Hour (24-hour format), left padded with zeros (00..23)
H		Hour (24-hour format), non-padded (0..23)
g	Yes	Hour (12-hour format), left padded with zeros (01..12)
G		Hour (12-hour format), non-padded (1..12)
j	Yes	Day of year, left padded with zeros (001..366)
J		Day of year, non-padded (1..366)
m	Yes	Month, left padded with zeros (01..12)
M		Month, non-padded (1..12)
i	Yes	Minute, left padded with zeros (01..59)

l		Minute, non-padded (1..59)
p	Yes	am/pm designation (lowercase)
P		AM/PM designation (uppercase)
s	Yes	Seconds, left padded with zeros (00..59)
S		Seconds, non-padded (0..59)
u	Yes	Week number of year with Sunday as first day of week, left padded with zeros (01..52)
U		Week number of year with Sunday as first day of week, non-padded (1..52)
v	Yes	Week number of year with Monday as first day of week, left padded with zeros (01..52)
V		Week number of year with Monday as first day of week, non-padded (1..52)
w	Yes	Day of week with Sunday as first day of week (0=Sunday) (0..6)
W	Yes	Day of week with Monday as first day of week (0=Monday) (0..6)

y	Yes	Year (1900..2100)
Y	Yes	Year, using 2-digit notation (00..99) When used as an input mask, forms tries to reasonably guess the century. (TODO: Elaborate)

Predefined literals: "/-.:, "

Examples: 01/01/2001: "m/d/y" Friday, June 1, 2001: "A, b d, Y"

Formatting Text Fields

TODO

Dropdown Fields

TODO

Check boxes

TODO

A Brief Introduction to Python

While GNUe Forms will eventually support a plethora of scripting languages, the default, and best-supported, language will always be Python. Python is a .....

If you do not know Python, don't worry! Python is one of the simplest languages to pick up. .....

Once multi-language support is added, the developer will be able to write triggers in Python, Perl, Ruby, Scheme, or possibly even Basic. .....

While Python is easy to learn, this section assumes that you know at least one programming language. It is beyond the scope of this guide to cover basic programming concepts. There are several excellent Python tutorials for those beginning programming available on the Web. Go to http://www.python.org/doc/ for a listing of available tutorials. They have docs for every stage of python programming, from new-to-programming to seasoned veteran.

The Basics

The first thing most people notice about Python is its reliance on whitespace for grouping.

Variables and Expressions

```
x = 1
```

Control Structures

```
if x == 1:
  print "Yip"


for f in (1,2,3):
  print f


for f in range(4):
  print f


n = 1
while n < 10:
  n += 1
```

Tuples, Lists, and Dictionaries... oh, my!

TODO

Exploring Trigger Namespaces

Introduction

TODO

Global Names

GNUe Forms supports the Python `global` construct, which can be used by the developer to define global variables and methods, or import modules globally. For example, assume the following code chunk is a form's `On-Startup` trigger:

```
##
## On-Startup [Form]
##

# We want to give our other triggers
# access to these three objects.
global math, myfunc, DEBUG

# We will use the math module a lot
# in our other triggers
import math

# A handy function
def myfunc(n1,n2):
  return n1+n2

# Are we in DEBUG mode?
# Enquiring triggers want to know...
DEBUG = 1

# This is an example of a non-global name.
# Only our On-Startup trigger sees this.
test = 2
```

Because Forms executes `On-Startup` before any other triggers, all other triggers within this form can now see `math`, `myfunc`, and `DEBUG`.

For example, an `On-Change` trigger could now do:

```
##
## On-Change (MyEntry)
##
if DEBUG:
  print "Starting value: %s" % self.get()
```

```
computed = myfunc(self.get(), 12)

AnotherEntry.set(math.floor(computed))
```

Note that if another trigger wanted to globally change the values of `math`, `myfunc`, or `DEBUG`, they would also have to use the `global` construct. The following section of code would only change `DEBUG` for this single execution of `On-Change`:

```
##
## On-Change (MyEntry)
##
DEBUG = 0

# ... other code ...
```

The `other code` in this example would see `DEBUG` as being `0`, but once the trigger was completed, `DEBUG` would return to being `1` for all future triggers. Now, suppose the trigger had instead looked like:

```
##
## On-Change (MyEntry)
##
global DEBUG
DEBUG = 0

# ... other code ...
```

Now, this trigger and all future triggers will see `DEBUG` as `0`.

Object Heirarchy

TODO

Fields and Entries

TODO

Blocks

TODO

Datasources

Datasource objects

The datasource objects, in addition to providing sensible methods for data access and manipulation, also act as python container objects whenever possible. For example, a

recordset can behave as a python dictionary, and resultsets behave as tuples. Any container-related behavior will be discussed after the supported methods are detailed.

Datasource

ResultSet

RecordSet

Examples

TODO

```
resultset = MyDataSource.createResultSet({'id':1})
recordset = resultset.nextRecord()
if recordset:
  DescrField.set(recordset['description'])
```

TODO

```
resultset = MyDataSource.createResultSet()
recordset = resultset.createRecord()
recordset['id'] = 1
recordset['description'] = 'Test Record'
recordset['amount'] = 10.00
```

TODO

```
resultset = MyDataSource.createResultSet({'id':1})
recordset = resultset.nextRecord()

# Delete the last record
recordset.deleteRecord()
```

TODO

```
# Query all records where amount is 5.00
resultset = MyDataSource.createResultSet({'amount':5})
for recordset in resultset:
  recordset['queue'] = 'Y'
```

TODO

Form

Creating and Using Libraries

Overview

TODO

Integration with GNUe Tools

Running Reports from Forms

TODO

Running Forms from Navigator

TODO

Advanced Topics

This section describes advanced forms concepts. [TODO: EXPAND]

Runtime Parameters

Forms supports runtime parameters that can be passed to a form instance at startup. Parameters are mainly useful for specifying conditions in datasources, but can also be accessed via triggers. This allows the perceived behavior of a form to be altered only by passing a parameter.

A good example is a form designed to service two divisions of a company. While you could offer an opening dialog that asks the user which division he wants to work on, an alternative is to modify his startup script to tell the form which division he works with. This especially works well when the worker only belongs in one division and will never need access to any others.

Note that in the above example, though, parameters are not a good substitute for access security. This example would strictly be for convenience, not security.

Parameters must be defined, with a default value, using the `<parameter>` tag:

```
<form>
  <parameter name="division" default="101"/>
  ...
</form>
```

Once defined, parameters can be passed to forms in one of two ways. The first is via the command line. Parameters can be passed in the format `parameter=value` on the command line appearing after the name of the form. For example:

```
gnue-forms myform.gfd division=101
```

Alternately, if the form is being called from another form, the trigger would look like:

```
## Run "myform.gfd"
form.runForm( 'myform.gfd', { 'division' : 101 } )
```

That is, the parameters would be passed to runForm as a Python dictionary. Once passed to the form, parameters can be used in one of two ways: via trigger code or as a parameter to a datasource condition.

First, triggers can access parameters using the form.getParameter() method. This method takes one argument, the case-insensitive parameter name. It returns the requsested parameter, or the default value if no parameter was passed on startup.

```
## Get the "company" parameter
division = form.getParameter('division')
```

Conditions are also a good place for parameters. Take the following fragment:

```
<datasource name="dtsExsample" table="sales"
            connection="sales"/>
  <condition>
    <eq>
      <cfield name="division"/>
      <cparam name="division"/>
    </eq>
  </condition>
</datasource>
```

With this in place, whenever the table `sales` is queried, the only records returned are the ones where the field `division` matches the parameter `division`. Note that if this datasource will also be used for inserting new rows, a `Pre-Insert` trigger is needed to set the `division` field:

```
##
## Pre-Insert [SalesBlock]
DivisionEntry.set(form.getParameter('division'))
```

External Python Modules

Python triggers have full access to your installed Python modules. For example, if your project needs the `twofish` cryptographic module, you can install it normally and do an `import twofish` in your triggers.

Alternately, GNUe's `gnue.conf` file supports an `ImportPath` directive. You can have this point to a directory containing your custom python modules.

Designing for Multiple Interfaces

A form definition, when designed within reasonable guidelines, can be run on a plethora of system architectures and a wide variety of user interfaces. By using the approach taken in this guide, most of your forms will, by default, run on a graphical workstation (X11, Windows, Mac), in a text-based session (telnet or ssh), or via a web browser (HTML). This section highlights a few key compatibility issues.

This list, while not exhaustive, should give you a good idea of common portability pitfalls. As with all things in GNUe, you will always have a choice on how to implement your application. GNUe is not about forcing rules on developers, but about providing viable options. There will be instances where the following suggestions simply are not feasible or practical. In any event, these are simply suggestions on getting the most out of Forms.

- **Images**
  Do not make your application dependent on displayed images. It would be acceptable, and appropriate, to display pictures for informational purposes. For example, when doing parts lookups, it would be appropriate to display a picture of the part for reference use. However, it would be normally be inappropriate to prevent the form from working if this image could not be displayed. (TODO: Better example?)

- **OS-specific trigger code**
  Python, the default trigger language, provides an extensive library of cross-platform functions. For example, it provides a library of file-access routines that work on all its supported platforms. This is really a broad category as trigger code has all the power of Python behind it.
- **Custom widgets**
  It is often tempting to use a new whiz-bang widget available on a certain platform/widget set. This will surely make your application hard to migrate to other platforms/interfaces, as well as restrict your ability to upgrade to a newer Forms version. Form's widget-set was carefully selected to be as multi-platform-friendly as possible, while still providing all the functionality most forms will need. If your application widgets are not supported by forms, there's a good chance that your form could be more functional with a slight rethinking of its design. Remember: a goal of Forms is to be usable on as many platforms as possible, not to exploit all the features of a particular platform.

Trigger Recipes

Over the coarse of writing a complex application, you will encounter a few situations where you will need a trigger to perform a common task. This section lists several

Timestamping a Record prior to a Commit

To automatically fill an entry with a timestamp retrieved from the database, you can use the datasource extension getTimestamp(). create a `Pre-Commit` trigger on that block. For example,

```
##
## Pre-Commit [MyBlock]
##

self.MyTimeField.set(MyDS.extensions.getTimestamp())
```

This example assumes your entry is named `MyTimeField and` your datasource is called `MyDS`.

As noted elsewhere in this guide, `Pre-Commit` is run prior to saving changes to the database regardless of whether the record in question is being inserted, updated, or deleted. If you want to timestamp only new records, you can use the same code listed above, only inside a `Pre-Insert` trigger. Similarly, if you only want to timestamp modifications, you can use a `Pre-Update` trigger.

At this point, you may be asking why did we go through `MyDS` to get to a database timestamp. After all, `MyDS` corresponds to a table, not to a database. [TODO: PROVIDE AN EXPLANATION]

By using a timestamp retrieved from the database server, you do not have to worry about differences in the client machines' times. If you would prefer to have the client's time, you can use python's time module.

Stamping a Record with User's Login prior to a Commit

To automatically fill an entry with a the user's login name, you can use form.getAuthenticatedUser() and creating a `Pre-Insert` trigger on that block. For example,

```
##
## Pre-Insert [MyBlock]
##

self.CreatedBy.set(form.getAuthenticatedUser())
```

This example assumes your entry is named `CreatedBy`. As noted elsewhere in this guide, `Pre-Insert` is run prior to saving a new record to the database.

This method is commonly called alongside the timestamping recipe above. Together, a Pre-Insert trigger to stamp a new record might look like:

```
##
## Pre-Insert [MyBlock]
##

self.CreatedBy.set(form.getAuthenticatedUser())
self.CreatedOn.set(MyDS.extensions.getTimestamp())
```

See the recipe for timestamping for more information on usage of `getTimestamp()`.

Auto-Populating an Entry from a Sequence

To automatically fill an entry with a value from a sequence, you can create a Pre-Insert trigger on that entry. For example,

```
##
## Pre-Insert [MyEntry]
##

self.set(MyDS.extensions.getSequence("MySequence"))
```

This example assumes your entry is named `MyEntry`, your datasource is called `MyDS`, and the sequence name as stored in the database is `MySequence`. Note that `MyEntry` and `MyDS` are both names originating in your form, whereas `MySequence` is a name originating in your database.

At this point, you may be asking why did we go through `MyDS` to get to a database sequence. After all, `MyDS` corresponds to a table, not to a database. [TODO: PROVIDE AN EXPLANATION]

Appendix A: Trigger Hierarchy

Forms supports

Appendix B: Form Elements

[TODO: Introduction]

Form Tags

form

No description provided

Attributes

Attribute	*Values*	*Default*	*Description*
name	*text*		A unique ID for the form.
readonly	`Y`, `N`	`N`	If set to `Y`, then no modifications to data by the end user will be allowed. The form will become a query-only form.
style	`dialog`		No description provided
title	*text*	`Untitled Form`	The title of the form.

Child Nodes

`connection`, `datasource`, `dialog`, `import-datasource`, `import-dialog`, `import-layout`, `import-logic`, `import-trigger`, `layout`, `logic`, `menu`, `options`, `parameter`, `trigger`

Connection Tags

connection

No description provided

Attributes

Attribute	*Values*	*Default*	*Description*
name	*text*		No description provided
provider	*text*		No description provided
comment	*text*		No description provided
dbname	*text*		No description provided
host	*text*		No description provided
service	*text*		No description provided

Datasource Tags

datasource

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided
cache	*number*	5	No description provided
connection	*text*		No description provided
detaillink	*text*		No description provided
distinct	Y, N	N	No description provided
explicitfields	*text*		No description provided
master	*text*		No description provided
masterlink	*text*		No description provided
order_by	*text*		No description provided
prequery	Y, N	N	No description provided
primarykey	*text*		No description provided
table	*text*		No description provided
type	*text*	object	No description provided

Child Nodes

condition, staticset

add

No description provided

Child Nodes

add, cconst, cfield, cparam, div, mul, sub

and

No description provided

Child Nodes

and, between, conditions, eq, ge, gt, le, like, lt, ne, negate, not, notbetween, notlike, notnull, null, or

between

No description provided

Child Nodes

add, cconst, cfield, cparam, div, mul, sub

cconst

No description provided

Attributes

Attribute	Values	Default	Description
value	*text*		No description provided

cfield

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided

condition

No description provided

Child Nodes

`and`, `between`, `eq`, `ge`, `gt`, `le`, `like`, `lt`, `ne`, `negate`, `not`, `notbetween`, `notlike`, `notnull`, `null`, `or`

cparam

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		No description provided

div

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

eq

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

ge

No description provided

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

gt

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

le

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

like

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

lt

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

mul

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

ne

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

negate

NO DESCRIPTION PROVIDED

Child Nodes

and, between, conditions, eq, ge, gt, le, like, lt, ne, negate, not, notbetween, notlike, or

not

NO DESCRIPTION PROVIDED

Child Nodes

and, between, conditions, eq, ge, gt, le, like, lt, ne, negate, not, notbetween, notlike, notnull, null, or

notbetween

NO DESCRIPTION PROVIDED

Child Nodes

add, cconst, cfield, cparam, div, mul, sub

notlike

NO DESCRIPTION PROVIDED

Child Nodes

add, cconst, cfield, cparam, div, mul, sub

notnull

NO DESCRIPTION PROVIDED

null

NO DESCRIPTION PROVIDED

or

NO DESCRIPTION PROVIDED

Child Nodes

and, between, conditions, eq, ge, gt, le, like, lt, ne, negate, not, notbetween, notlike, notnull, null, or

staticset

NO DESCRIPTION PROVIDED

Attributes

Attribute	*Values*	*Default*	*Description*
fields	*text*		NO DESCRIPTION PROVIDED

Child Nodes

`staticsetrow`

staticsetfield

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	*text*		NO DESCRIPTION PROVIDED
value	*text*		NO DESCRIPTION PROVIDED

staticsetrow

NO DESCRIPTION PROVIDED

Child Nodes

`staticsetfield`

sub

NO DESCRIPTION PROVIDED

Child Nodes

`add`, `cconst`, `cfield`, `cparam`, `div`, `mul`, `sub`

Dialog Tags

dialog

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	*text*		A unique ID for the form.
readonly	`Y`, `N`	`N`	If set to `Y`, then no modifications to data by the end user will be allowed. The form will become a query-only form.
style	`dialog`	`dialog`	NO DESCRIPTION PROVIDED
title	*text*	`Untitled Form`	The title of the form.

Layout Tags

layout

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
tabbed	`bottom`, `left`, `right`, `top`		Allows a form to convert it's pages as notebook tabs. Allowed values are `left`, `right`, `bottom`, `top`.

Child Nodes

`import-page`, `page`

box

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
focusorder	*number*		NO DESCRIPTION PROVIDED
label	*text*		An optional text label that will be displayed on the border.
name	*text*		NO DESCRIPTION PROVIDED

button

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
focusorder	*number*		NO DESCRIPTION PROVIDED
label	*text*		The text that should appear on the button
name	*text*		A unique ID for the widget. Useful for importable buttons.

entry

An `entry` is the visual counterpart to a `field`.

Attributes

Attribute	Values	Default	Description
block	*text*		The name of the block that this ties to.
field	*text*		The name of the field that this ties to.
focusorder	*number*		NO DESCRIPTION PROVIDED
hidden	`Y`, `N`	`N`	If defined the entry widget will not be displayed on the form. This is usefull for fields the user doesn't need to know about that you wish to update via triggers.
name	*text*		The unique ID of the entry.

Attribute	Values	Default	Description
navigable	`Y`, `N`	`Y`	It false, the user will be unable to navigate to this entry. Triggers can still alter the value.
rowSpacer	*number*		NO DESCRIPTION PROVIDED
rows	*number*		NO DESCRIPTION PROVIDED
style	`checkbox`, `default`, `dropdown`, `label`, `password`	`default`	The style of entry widget requested. Currently either `text`, `label`, `checkbox`, or `dropdown`. To use `dropdown` you are required to use both the `fk_source`, `fk_key`, and `fk_description` attributes. The `label` style implies the `readonly` attribute.

image

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
block	*text*		The name of the block that this ties to.
field	*text*		The name of the field that this ties to.
focusorder	*number*		NO DESCRIPTION PROVIDED
name	*text*		NO DESCRIPTION PROVIDED
type	`BINARY`, `URL`	`URL`	NO DESCRIPTION PROVIDED

label

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
text	*text*		The text to be displayed.
alignment	`center`, `left`, `right`	`left`	The justification of the label. Can be one of the following: `left`, `right`, or `center`. Requires that the `width` attribute be set.
name	*text*		The unique ID of the label.
rowSpacer	*number*		Overriders the rowSpace setting defined at the block level.
rows	*number*		Overrides the rows setting defined at the block level.

page

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
caption	*text*		For tabbed or popup pages, this contains the caption to use for the page.
name	*text*		A unique ID for the widget. This is only useful when importing pages from a library.
style	`normal`	`normal`	The type of page when importing pages from a library.
transparent	`Y`, `N`	`N`	If set, then you can tab out of the page via next- or previous-field events. Makes navigation in mutlipage forms easier. If false, focus stays within a page until user explicitly moves to another page

Child Nodes

`box`, `button`, `entry`, `image`, `import-button`, `label`, `scrollbar`

scrollbar

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
block	*text*		The `block` to which this scrollbar scrolls.

Logic Tags

logic

NO DESCRIPTION PROVIDED

Child Nodes

`block`, `import-block`

block

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	*text*		A unique ID for the widget.The name of the widget. No blocks can share the same name without causing namespace collisions in user triggers.
datasource	*text*		The name of a datasource (defined in by a `<datasource>` tag.) that provides this block with it's data.

Attribute	Values	Default	Description
restrictDelete	`Y`, `N`	`N`	If set then the user will be unable to request that a record be deleted via the user interface.
restrictInsert	`Y`, `N`	`N`	If set then the user will be unable to request that new records be inserted into the block.
rowSpacer	*number*		Adjusts the vertical gap of this number of rows between duplicated widgets. Serves the same purpose as some of the gap attributes on individual widgets.
rows	*number*		Any widgets inside the block will display this number of copies in a verticle column. Simulates a grid entry system.
transparent	`Y`, `N`	`Y`	If set, then you can tab out of the block via next- or previous-field events. Makes navigation in mutliblock forms easier. If false, focus stays within a block until user explicitly moves to another block

Child Nodes

`field`, `import-field`

field

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	*text*		The unique ID of the entry. Referenced in master/detail setups as well as triggers.
case	`lower`, `mixed`, `upper`	`mixed`	NO DESCRIPTION PROVIDED
default	*text*		The default value for any new records created. If the field is visible the user can override the value.
displaymask	*text*		NO DESCRIPTION PROVIDED
editOnNull	`Y`, `N`	`N`	NO DESCRIPTION PROVIDED
field	*text*		The name of the field in the datasource to which this widget is tied.
fk_description	*text*		NO DESCRIPTION PROVIDED
fk_key	*text*		NO DESCRIPTION PROVIDED
fk_refresh	`change`, `commit`, `startup`	`startup`	NO DESCRIPTION PROVIDED
fk_source	*text*		NO DESCRIPTION PROVIDED

Attribute	Values	Default	Description
formatmask	*text*		NO DESCRIPTION PROVIDED
ignoreCaseOnQuery	`Y`, `N`	`N`	If defined the entry widget ignores the case of the information entered into the query mask.
inputmask	*text*		NO DESCRIPTION PROVIDED
ltrim	`Y`, `N`	`N`	Trim extraneous space at beginning of user input.
max_length	*number*		The maximum number of characters the user is allowed to enter into the entry.
min_length	*number*	`0`	The minimum number of characters the user must enter into the entry.
queryDefault	*text*		The form will be populated with this value automatically when a query is requested. If the field is visible the user can still override the value.
readonly	`Y`, `N`	`N`	It defined the user will be unable to alter the contents of this entry. Triggers can still alter the value.
required	`Y`, `N`	`N`	This object cannot have an empty value prior to a commit.
rtrim	`Y`, `N`	`Y`	Trim extraneous space at end of user input.
sloppyQuery	*text*		When set, whatever value the user enters for the query mask is rewritten with % between each character. Thus `example` would be queried as `%e%x%a%m%p%l%e%`
typecast	`date`, `number`, `text`	`text`	The type of data the entry widget will accept. Possible values are `text`, `number`, `date`.
value	*text*		NO DESCRIPTION PROVIDED

Menu Tags

menu

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	*text*		NO DESCRIPTION PROVIDED
enabled	`Y`, `N`	`N`	NO DESCRIPTION PROVIDED
event	*text*		NO DESCRIPTION PROVIDED
label	*text*		NO DESCRIPTION PROVIDED
leader	*text*		NO DESCRIPTION PROVIDED

Attribute	Values	Default	Description
location	*text*		No description provided
trigger	*text*		No description provided
type	*text*		No description provided

Options Tags

options

No description provided

Child Nodes

`author`, `description`, `name`, `option`, `tip`, `title`, `version`

author

No description provided

Attributes

Attribute	Values	Default	Description
name	`author`	`author`	No description provided
value	*text*		No description provided

description

No description provided

Attributes

Attribute	Values	Default	Description
name	`description`	`description`	No description provided
value	*text*		No description provided

name

No description provided

Attributes

Attribute	Values	Default	Description
name	`name`	`name`	No description provided
value	*text*		No description provided

option

No description provided

Attributes

Attribute	Values	Default	Description
name	*text*		NO DESCRIPTION PROVIDED
value	*text*		NO DESCRIPTION PROVIDED

tip

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	`tip`	`tip`	NO DESCRIPTION PROVIDED
value	*text*		NO DESCRIPTION PROVIDED

version

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	`version`	`version`	NO DESCRIPTION PROVIDED
value	*text*		NO DESCRIPTION PROVIDED

Parameter Tags

parameter

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
name	*text*		NO DESCRIPTION PROVIDED
default	*text*		NO DESCRIPTION PROVIDED
description	*text*		NO DESCRIPTION PROVIDED
required	`Y`, `N`	`N`	NO DESCRIPTION PROVIDED
type	*text*	`char`	NO DESCRIPTION PROVIDED

Trigger Tags

trigger

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
language	`python`	`python`	NO DESCRIPTION PROVIDED
name	*text*		NO DESCRIPTION PROVIDED
src	*text*		NO DESCRIPTION PROVIDED
type	*text*		NO DESCRIPTION PROVIDED

Import Tags

import-block

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		NO DESCRIPTION PROVIDED
name	*text*		A unique ID for the widget.The name of the widget. No blocks can share the same name without causing namespace collisions in user triggers.
datasource	*text*		The name of a datasource (defined in by a `<datasource>` tag.) that provides this block with it's data.
restrictDelete	`Y`, `N`	`N`	If set then the user will be unable to request that a record be deleted via the user interface.
restrictInsert	`Y`, `N`	`N`	If set then the user will be unable to request that new records be inserted into the block.
rowSpacer	*number*		Adjusts the vertical gap of this number of rows between duplicated widgets. Serves the same purpose as some of the gap attributes on individual widgets.
rows	*number*		Any widgets inside the block will display this number of copies in a verticle column. Simulates a grid entry system.
transparent	`Y`, `N`	`Y`	If set, then you can tab out of the block via next- or previous-field events. Makes navigation in mutliblock forms easier. If false, focus stays within a block until user explicitly moves to another block

import-button

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
focusorder	*number*		No description provided
label	*text*		The text that should appear on the button
name	*text*		A unique ID for the widget. Useful for importable buttons.

import-datasource

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
name	*text*		No description provided
cache	*number*	`5`	No description provided
connection	*text*		No description provided
detaillink	*text*		No description provided
distinct	`Y`, `N`	`N`	No description provided
explicitfields	*text*		No description provided
master	*text*		No description provided
masterlink	*text*		No description provided
order_by	*text*		No description provided
prequery	`Y`, `N`	`N`	No description provided
primarykey	*text*		No description provided
table	*text*		No description provided
type	*text*	`object`	No description provided

import-dialog

No description provided

Attributes

Attribute	Values	Default	Description
library	*text*		No description provided
name	*text*		A unique ID for the form.
readonly	`Y`, `N`	`N`	If set to `Y`, then no modifications to data by the end user will be allowed. The form will become a query-only form.
style	`dialog`	`dialog`	No description provided

Attribute	Values	Default	Description
title	*text*	`Untitled Form`	The title of the form.

import-field

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		NO DESCRIPTION PROVIDED
name	*text*		The unique ID of the entry. Referenced in master/detail setups as well as triggers.
case	`lower`, `mixed`, `upper`	`mixed`	NO DESCRIPTION PROVIDED
default	*text*		The default value for any new records created. If the field is visible the user can override the value.
displaymask	*text*		NO DESCRIPTION PROVIDED
editOnNull	`Y`, `N`	`N`	NO DESCRIPTION PROVIDED
field	*text*		The name of the field in the datasource to which this widget is tied.
fk_description	*text*		NO DESCRIPTION PROVIDED
fk_key	*text*		NO DESCRIPTION PROVIDED
fk_refresh	`change`, `commit`, `startup`	`startup`	NO DESCRIPTION PROVIDED
fk_source	*text*		NO DESCRIPTION PROVIDED
formatmask	*text*		NO DESCRIPTION PROVIDED
ignoreCaseOnQuery	`Y`, `N`	`N`	If defined the entry widget ignores the case of the information entered into the query mask.
inputmask	*text*		NO DESCRIPTION PROVIDED
ltrim	`Y`, `N`	`N`	Trim extraneous space at beginning of user input.
max_length	*number*		The maximum number of characters the user is allowed to enter into the entry.
min_length	*number*	`0`	The minimum number of characters the user must enter into the entry.
queryDefault	*text*		The form will be populated with this value automatically when a query is requested. If the field is visible the user can still override the value.

Attribute	Values	Default	Description
readonly	`Y`, `N`	`N`	It defined the user will be unable to alter the contents of this entry. Triggers can still alter the value.
required	`Y`, `N`	`N`	This object cannot have an empty value prior to a commit.
rtrim	`Y`, `N`	`Y`	Trim extraneous space at end of user input.
sloppyQuery	*text*		When set, whatever value the user enters for the query mask is rewritten with % between each character. Thus `example` would be queried as `%e%x%a%m%p%l%e%`
typecast	`date`, `number`, `text`	`text`	The type of data the entry widget will accept. Possible values are `text`, `number`, `date`.
value	*text*		NO DESCRIPTION PROVIDED

import-layout

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		NO DESCRIPTION PROVIDED
tabbed	`bottom`, `left`, `right`, `top`		Allows a form to convert it's pages as notebook tabs. Allowed values are `left`, `right`, `bottom`, `top`.

import-logic

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		NO DESCRIPTION PROVIDED

import-page

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		NO DESCRIPTION PROVIDED
caption	*text*		For tabbed or popup pages, this contains the caption to use for the page.

Attribute	Values	Default	Description
name	*text*		A unique ID for the widget. This is only useful when importing pages from a library.
style	`normal`	`normal`	The type of page when importing pages from a library.
transparent	`Y`, `N`	`N`	If set, then you can tab out of the page via next- or previous-field events. Makes navigation in mutlipage forms easier. If false, focus stays within a page until user explicitly moves to another page

import-trigger

NO DESCRIPTION PROVIDED

Attributes

Attribute	Values	Default	Description
library	*text*		NO DESCRIPTION PROVIDED
language	`python`	`python`	NO DESCRIPTION PROVIDED
name	*text*		NO DESCRIPTION PROVIDED
src	*text*		NO DESCRIPTION PROVIDED
type	*text*		NO DESCRIPTION PROVIDED

Appendix C: Form Objects

TODO

Form

getParameter()

Syntax:

`getParameter(`*parameter*`)`

Description:

Returns a runtime parameter, or the default value of such if the user did not pass the requested runtime parameter. See **Runtime Parameters** on page 31 for more information.

Example:

```
# Get the runtime parameter "company"
company = form.getParameter("company")
```

setParameter()

Syntax:

`setParameter(`*parameter*`,` *value*`)`

Description:

Changes the value of a runtime parameter. See **Runtime Parameters** on page 31 for more information.

Example:

```
# Set the runtime parameter "company" to "101"
form.setParameter("company","101")
```

getFocus()

Syntax:

`getFocus(`*object*`)`

Description:

Request that the current focus be given to `object`. If `object` is a block or a page, then focus will be given to the first navigable entry on that page or block. All appropriate `Pre-FocusOut`, `Pre-FocusIn`, `Post-FocusOut`, and `Post-FocusIn` triggers will be

executed. If `setFocus` is called on a non-navigable item (such as a label), the call is ignored and focus does not change.

Example:

```
# Request that MyEntry gets the current focus
form.setFocus(MyEntry)
```

setFocus()

Syntax:

`setFocus(`*object*`)`

Description:

Request that the current focus be given to `object`. If `object` is a block or a page, then focus will be given to the first navigable entry on that page or block. All appropriate `Pre-FocusOut`, `Pre-FocusIn`, `Post-FocusOut`, and `Post-FocusIn` triggers will be executed. If `setFocus` is called on a non-navigable item (such as a label), the call is ignored and focus does not change.

Example:

```
# Request that MyEntry gets the current focus
form.setFocus(MyEntry)
```

setStatusText()

Syntax:

`setStatusText(`*text*`)`

Description:

For user interfaces that support a status bar, or some textual equivalent, set the displayed text. For interfaces without a status bar equivalent, this function is meaningless.

Example:

```
# Tell the user how great they are
form.setStatusText("Dude, you are the best user ever!")
```

showMessage()

Syntax:

`showMessage(`*text*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

commit()

Syntax:

```
commit()
```

Description:

Description goes here.

Example:

```
# Code Sample
```

close()

Syntax:

```
close()
```

Description:

Description goes here.

Example:

```
# Exit the current form
form.close()
```

getAuthenticatedUser()

Syntax:

```
getAuthenticatedUser([connection])
```

Description:

Description goes here.

Example:

```
# Set "modified_by" to the current user's login
modified_by.set(form.getAuthenticatedUser())
```

Datasource

createResultSet()

Syntax:

`createResultSet([`*conditions*`], [`*readOnly*`] )`

Description:

Description goes here.

Example:

```
# Code Sample
```

simpleQuery()

Syntax:

`simpleQuery(`*dictionary*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

delete()

Syntax:

`delete()`

Description:

Description goes here.

Example:

```
# Code Sample
```

Block

clear()

Syntax:

```
clear()
```

Description:

Clears the current block with an empty result set.

Example:

```
# Clear out MyBlock
MyBlock.clear()
```

gotoRecord()

Syntax:

`gotoRecord(`*index*`)`

Description:

Move to the record indicated by `index`. If `index` is negative, then move relative to the last record. Records are numbered beginning with `0`.

Example:

```
# Go to the second record in this block
MyBlock.gotoRecord(1)

# Go to the last record in this block
MyBlock.gotoRecord(-1)
```

newRecord()

Syntax:

```
newRecord()
```

Description:

Inserts a new record immediately following the current record. This new record will then become the current record. The `On-NewRecord` trigger is executed for the newly created record and any default values are recorded.

Example:

```
# Code Sample
MyBlock.newRecord()
```

nextRecord()

Syntax:

```
nextRecord()
```

Description:

Navigate to the next record. If the block is currently on the last record, then this method returns `0` (false). Otherwise it returns `1` (true).

Example:

```
# Move to the next record
MyBlock.nextRecord()
```

prevRecord()

Syntax:

```
prevRecord()
```

Description:

Navigate to the previous record. If the block is currently on the first record, then this method returns `0` (false). Otherwise it returns `1` (true).

Example:

```
# Move to the previous record
MyBlock.prevRecord()
```

deleteRecord()

Syntax:

```
deleteRecord()
```

Description:

Mark the current record as *deleted*. On the next save, this record will be permanently removed.

Example:

```
# Code Sample
MyBlock.deleteRecord()
```

parent

Description:

This read-only property contains the parent container of this block. The parent container is usually a page.

Example:

```
# Get MyBlock's parent page
page = MyBlock.parent
```

Entry

allowedValues()

Syntax:

`allowedValues()`

Description:

Returns a tuple containing valid values for this entry. This call will only return a set when a `fk_source` has been specified for the entry.

Example:

```
# Code Sample
if 'Test' not in MyEntry.allowedValues():
  MyEntry.set(None)
```

autofillBySequence()

Syntax:

`autofillBySequence(`*sequence*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

isEmpty()

Syntax:

`isEmpty()`

Description:

Returns true if the current entry is considered empty. Empty is usually associated with a blank, or null, value.

Example:

```
# Set MyEntry to 0 if it has no other value.
if MyEntry.isEmpty():
  MyEntry.set(0)
```

set()

Syntax:

`set(`*value*`)`

Description:

Description goes here.

Example:

```
# Code Sample
```

get()

Syntax:

`get()`

Description:

Description goes here.

Example:

```
# Code Sample
```

resetForeignKey()

Syntax:

```
resetForeignKey()
```

Description:

Description goes here.

Example:

```
# Code Sample
```

parent

Description:

This read-only property contains the parent container of this entry. The parent container will typically be a block, unless container boxes are used.

Example:

```
# Get this entry's parent block
block = self.parent
```

readonly

Description:

Description goes here.

Example:

```
# Set MyEntry to be readonly if not already
if not MyEntry.readonly:
  MyEntry.readonly = 1
```

Appendix D: Data Objects

TODO

Result Set

xxxx()

Syntax:

`getParameter(`*parameter*`)`

Description:

Description....

Example:

```
# Get the runtime parameter "company"
company = form.getParameter("company")
```

Record Set

xxxx()

Syntax:

`getParameter(`*parameter*`)`

Description:

Description....

Example:

```
# Get the runtime parameter "company"
company = form.getParameter("company")
```

Appendix E: Sample Librarian Schema

[TODO: Add example schema explanations]

Appendix F: Glossary

database

datasource

entry

field

python

Appendix G: GNU Free Documentation License

GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000, 2001, 2002 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some wordprocessors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical

measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

c) State on the Title page the name of the publisher of the Modified Version, as the publisher.

d) Preserve all the copyright notices of the Document.

e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

h) Include an unaltered copy of this License.

i) Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

k) For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

m) Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

n) Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

o) Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the

same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document,

and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Alphabetical Index

C
CUSTOM WIDGETSustom widgets 33

D
DATASOURCEatasource 10-13, 35, 36
DESIGNEResigner 7, 10-12

G
GETSEQUENCEetSequence 36
GETTIMESTAMPetTimestamp 35

P
PRE-COMMITre-Commit 35
PRE-INSERTre-Insert 35, 36
PYTHONython 7, 32-34

S
SEQUENCEequence 36

T
TIMESTAMPimestamp 35

X
XMLml 12, 13