FLYING LOGIC
Scripting Guide
Version 1.7

Sciral

Sciral
Glendora, CA 91740
FlyingLogic.com

Contents

Introduction

Flying Logic implements a powerful internal scripting language. Scripts can be used to create or manipulate documents and change Flying Logic's application preferences. In addition, document importers and exporters can be written as scripts, including document transformation through Extensible Stylesheet Language Transformations (XSLT).

Scripting with Python

Flying Logic scripts are written in the Python™ programming language. This guide does not provide an extensive tutorial on Python, but only a quick overview for simple understanding of writing scripts. Online sources for information on Python or the specific implementation Flying Logic uses called Jython (Python for the Java Platform) can be founded in "Online Resources" on page 13.

Python is an interpreted object-oriented program language. Small programs (called scripts) can be written and executed by Flying Logic via the **Run Script**, **Import Via Script** and **Export Via Script** menu items.

Simple scripts can be written without a knowledge of object-oriented programming, but complex scripts can benefit from such knowledge. Most of the variables that Flying Logic provided to scripts are objects, but can be manipulated fairly easily.

Here is a very simple script:

```
Application.alert("Hello, world!")
```

If you type the line above into a text file named "hello.py" and then select that file via the Run Script menu item, a dialog will appear with the message "Hello, world!" The ".py" part of the file name is the standard file extension for Python scripts.

Every Python script run via Flying Logic is provided a number of predefined variables. One of those is the Application object, which provides access to various features of the application that are not document-specific. The *alert* method of Application displays the string of text provided as a parameter to the method.

A second variable provided to all scripts is the Document object. Here is a simple example of using this object:

```
Application.alert( document.title )
```

Create a script with the above line. In Flying Logic create a new document, open the Document Inspector and enter a value into the Title field. Use the Run Script command to execute the new script. A dialog should open with the "message" being the title you entered.

The *document object* gives you access to everything in the "current document"; i.e., the document that was active when you ran the script. The *title* instance variable of this document object gives access to the title field of the Document Inspector.

Many variables in the document (and Application) object can be read and written. If you run a script with the following line:

```
document.title = "A Simple Script Example"
```

you will find that the title field of the document has been changed.

Here is one more simple example that also demonstrates how to create conditional expressions, loops and blocks in Python and shows how to use the output console. Create a script with the following lines:

```
for ge in document.selection:
    if ge.isEntity:
        print ge.title
print "Done"
```

Python does not indicate the end of a statement with a semicolon or other delimiter like many other languages. Instead a newline indicates the end of a statement. The "print element.title" line is statement.

There are two looping statements in Python. The *for* loop executes a block once for each element in a sequence. A block in Python is indicated by indentation. The lines if **ge.isEntity:** and **print ge.title** are inside the for block, while **print "Done"** is not. The colon at the end of the *for* statement indicates the start of a block.

The *selection* variable of *document* returns a sequence containing objects representing all the selected graph elements in the document. The each loop of for block, the local variable *ge* is assigned one of the elements of selection.

The first line in the *for* block is a conditional *if* statement. If the condition is true, the conditional block is executed. The line **print ge.title** is inside the *if* statement's block. (This also makes it a nested block.) If the

variable *isEntity* of the *ge* object is **true**, then the title of the *ge* object (which must represent and entity in Flying Logic) will be printed.

You may be wondering where the *print* function prints text (or *strings* as they are called in Python). By default strings are printed to Flying Logic's *scripting console*, which is a window that will open to display printed strings. This console will also appear when an exception occurs in the script and an error message is printed.

In summary, the above script prints the title of all selected entities to the console and then the string "Done".

The Application and document objects have many variables and methods that can be used to perform virtually every command in Flying Logic. The document object is an instance of the Document class. A script can retrieve objects representing other open documents other that the "current document" or even open or create new documents via the Application object. The Application object is a *singleton*— it is an instance of the Application class, but there is only ever one instance of that class.

A complete class reference can be found in <u>"Pre-Defined Variables and Class Reference" on page 15</u>.

About Jython and Java

API 1.7 Flying Logic uses the 2.7.0 version of Jython, which is compatible with the 2.7 command-line version of Python; i.e., CPython. Scripts have access to all features of a vanilla installation of Jython.

Scripts also have access to all Java packages that are distributed with Java SE Release 7. Flying Logic is currently being migrated to Java SE Release 8, but at this time scripts should not depend on features only found in Release 8 or deprecated in Release 7.

<u>"Example CSV Importer" on page 81</u> has an example of directly accessing Java to implement dialogs for an importer.

Writing Importers and Exporters

Flying Logic document importers and exporters can be written in Python. The same pre-defined classes and variables exist for a regular

script, but there are special functions and variables that a script must define to function as an importer or exporter.

An importer must define the function:

```
def importDocument(filename):
    """ code here """
```

This function is called to import the file with the given filename. If you are importing into the current document, you can use the pre-defined global variable *document*. To instead create a new document, call the method *newDocument()* in the Application object.

An importer must also define a global variable with the identifier *importItemLabel*. This is used to determine the label for the menu item that will be assigned to the importer script. For example:

```
importItemLabel = "Import Diagram from CSV File"
```

An exporter must define the function

```
def exportDocument(filename):
    """ code here """
```

and a global variable with the identifier exportLabelLabel.

```
exportItemLabel = "Export Diagram to CSV File"
```

For an exporter you will normally be working the existing *document* object.

You can open, read and write files by using either the Python file functions supported by Jython or the standard file classes in Java SE.

XSLT Importers and Exporters

Flying Logic's Python interface can also be used to create importers and exporters that perform their functionality via Extensible Stylesheet Language Transformations (XSLT). XSLT is a language for transforming XML documents into other XML documents. The transformation is encoded in an XSLT input document, which is itself an XML document. These XSLT input documents can either be separate files or embedded in a Python script.

Since Flying Logic documents are in XML format, XSLT can be used to

transform other document formats to and from Flying Logic. Performing this transformation does require that a XSLT input document author know the schema of the XML in a Flying Logic document. Instructions for downloading the complete XML Schema Definition (XSD) for Flying Logic documents can be found in , followed by additional information on how to interpret the schema.

Here is a short example of how to import a document using a string as the XSLT document:

```
xslt_string = """ some XSLT document as embedded string """
def importDocument(filename):
    Application.importDocument( filename, \
(Application.XSLT_STRING, xslt_string) )
```

> *Note:* The \ character in Python code examples indicates *line continuation*, i.e., that the characters on the next line logically belong with characters on the line where the line continuation symbol appears.

More detailed example of an importer and exporter can be found in .

Using Third-Party Python and Java Libraries

It is possible to access third-party Python and Java libraries from a script using the standard import feature of Python. To facilitate this, the global variable **scriptParentDirectory** is initialized with the path to the script's location in the filesystem. A script can then be bundled with additional resources it requires, including libraries and other assets.

Use of a third-party Python library requires adding the path to the library to the Python import search path. This requires appending the library's directory to **sys.path**.

```
import sys
sys.path.append(scriptParentDirectory)
import some_python_package
```

Use of a third-party Java library requires calling the Application method appendClassPath. The parameter to this method should be a list of either directories containing Java class files or paths to jar files.

```
Application.appendClassPath( [scriptParentDirectory + "/
SomeJavaLibrary.jar"] )
import some_java_package
```

> *Note:* Using third-party JDBC libraries to access databases requires the use of one of two special methods found in the Application class: createCompatibleDriver or createSQLConnection. See those methodfs for examples of usage.

Scripting Console

The **print** function in Python outputs a string to standard output. A Python program running in Flying Logic has its standard out redirected to the Scripting Console window. This window will appear whenever the **print** function is used.
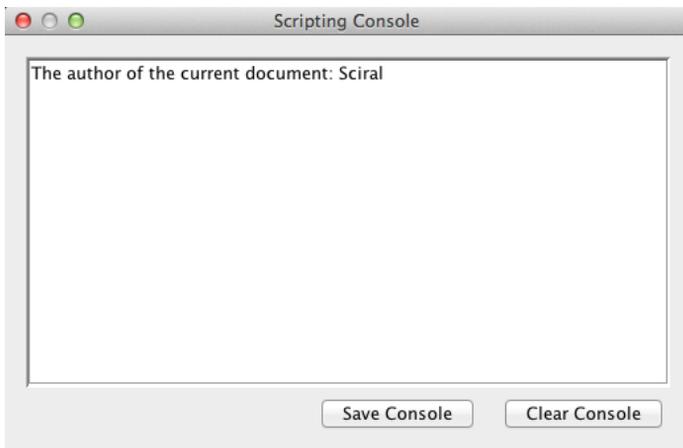
A multi-line text area shows every string printed. For example the text area above shows the result of the statement

```
print "The author of current document:",\
document.author
```

assuming the author field in the Document inspector has been set to "Sciral".

The text persists between execution of different scripts. Select the **Clear Console** button to erase the text. Select the **Save Console** button to save the text to a file.

If the Scripting Console is closed or hidden behind other windows, it can be made to appear by selecting the **Show Console** menu item.

Functionality Changes While Running a Script

There are some changes to Flying Logic's functionality while a script is running.

- Layout of changes to a graph are deferred until the script completes.

- Auto-editing of new entity titles is disabled.

- All changes to a document are coalesced into one undo action (but see the beginCoalesceUndo method of Document for how to control this feature).

Exception Handling

Some scripting methods can return a Java exception instead of a Python exception. To handle these Java exceptions, you can import the class java.lang.Exception and then catch them.

```
from java.lang import Exception
# assume you have variables referencing an edge and a group
try:
    document.modifyAttribute([anEdge], "parent", anGroup)
catch Exception, e:
    print e.message
```

Alternately, you can import other Java exception classes if you want to handle exceptions caused by calling Java library code directly.

Uncaught exceptions will be printed to the Scripting Console. This in-

cludes uncaught Java exceptions, which prevents the Flying Logic's normal exception report dialog from being displayed.

API Version

Features of the scripting API that appear in only particular versions or later will have a notation like: **API 1.7**.

Online Resources

Python™ Language and Jython

Python Programming Language – Official Website
http://www.python.org

The Jython Project
http://www.jython.org

Extensible Stylesheet Language Transformations (XSLT)

XSLT Tutorial
http://www.w3schools.com/XSL/

XSL Transformations (XSLT) Version 2.0
http://www.w3.org/TR/2007/REC-xslt20-20070123/

XSLT at Wikipedia
http://en.wikipedia.org/wiki/XSLT

XML Schema Definition Language (XSD)

XML Schema Tutorial
http://www.w3schools.com/schema/

XML Schema (W3C) at Wikipedia
http://en.wikipedia.org/wiki/XML_Schema_(W3C)

Other Standards

HTML Color Names
http://www.w3schools.com/html/html_colornames.asp

Pre-Defined Variables and Class Reference

Pre-Defined Variables

These variables are defined for each script and are the primary access to Flying Logic documents and other features.

document
> This represents the current document and is an instance of the Document class.

Application
> This is a singleton instance of the Application class. Methods and variables that are not related to particular documents can be accessed through this object.

scriptParentDirectory
> The path to the directory containing the script.

scriptMode
> Has the value "importer", "exporter" or "standard". This allows one script to act in any of the three modes; i.e., a script could be written as an importer and exporter.

Classes Overview

Class	Description
Document	an instance of a Flying Logic document
Application	the Flying Logic application
GraphElem	the base class of all graph element types
Edge	derived class of GraphElem for an edge
VertexElem	derived from GraphElem, base class for entity, junctor and group
Entity	derived class from VertexElem for an entity
Junctor	derived class from VertexElem for a junctor
Group	derived class from VertexElem for a group

Domain	a domain
EntityClass	an entity class in a domain
Symbol	a symbol
Color	an immutable RGB color value
Date	an immutable date
Resource	a human, capital, or production resource
Workweek	a workweek for a resource
GraphElemFilter	base class for filters for creating a list of GraphElem
FontSpec	a font specification
GraphOperator	base class for user-defined graph operators
TextEditor	class representing a modifiable text field
VertexOperator	base class for system-defined operators
CalendarEvent	an exception to the workweek
LayoutLine	position information for one line of a title or annotation **API 1.7**

Classes Detailed Reference

Document

The Document class represents a Flying Logic document. Every script has a global variable *document* that in an instance of Document representing the current document.

title
the title of the document, a string

author
the author of the document, a string

comment
the comments of the document, a string

keywords
the keywords of the document, a string

edgeWeightsVisible
the edge weights visible setting, either True of False

confidenceVisible
    the confidence visible setting, either True of False

annotationNumbersVisible
    the annotation numbers visible setting, either True of False

edgeAnnotationsVisible
    the edge annotations visible setting, either True of False

entityIDVisible
    the entity ID visible setting, either True of False

addEntityAsSuccessor
    the add entity as successor setting, either True of False

projectManagementVisible
    the project management visible setting, either True of False

layoutIncrementally
    the layout incrementally setting, either True of False

browseAnnotations
    the browse annotations setting, either True of False

printShowSelectionHalo
    the show selection halo when printing setting, either True of False

canvasView
    the current kind of canvas view, either graph (Application. GRAPH_VIEW) or chart (Application.CHART_VIEW)

startDate
    the start date of project as a Date instance when project management is enabled, otherwise None. This field is read-only if using finish-to-start scheduling, with the following exception. Setting this variables to None **disables** project management.

actualStartDate `API 1.7`
    the start date of the earliest task, which may be earlier than project start date because of the task having a preferred start or finish date. Read only

finishDate
    the finish date of project as a Date instance when project management is enabled, otherwise None. The field is read-only if

using finish-to-start scheduling, with the following exception. Setting this variable to None **disables** project management.

startTime
returns the number of hours into the work day that the project begins

finishTime
returns the number of hours into the work day that the project ends

standardCalendar
the standard calendar is the Workweek instance uses for all tasks not assigned a resource when project management is enabled, otherwise None. Note that this variable always returns a deep copy, not the current instance. Modifications will not be saved unless you set the copy

For example:

```
stdCalendar = document.standardCalendar
stdCalendar.workdays = Workweek.MONDAY_MASK ^ theWeek.workdays
document.standardCalendar = stdCalendar
```

scheduleBasis
the direction of project scheduling, either Application. SCHEDULE_FROM_START_DATE (the default) or Application. SCHEDULE_FROM_END_DATE

~~**workweek**~~
This field is deprecated. Use **standardCalendar** instead.

calendars
a list of all defined calendars. These calendars, which are Workweek instances, are copies. To make permanent changes call the method **updateCalendar**. List can be empty (read only).

resources
a list of all defined resources. These resources are copies. To make permanent changes call the method **updateResource**. List can be empty (read only)

leftHeader
the left header text, a string

middleHeader
>   the middle header text, a string

rightHeader
>   the right header text, a string

leftFooter
>   the left footer text, a string

middleFooter
>   the middle footer text, a string

rightFooter
>   the right footer text, a string

orientation
>   the orientation of graph (see possible values under Application: orientation types)

bias
>   the bias of graph (see possible values under Application: bias types)

defaultEntityClass
>   the default entity class, usually *Generic* (read only instance of EntityClass)

selectedEntityClass
>   the current entity class of newly-created entities when not explicitly specified, an instance of EntityClass

defaultJunctorOperator
>   the current operator of newly-created junctors, an instance of VertexOperator

entityOperator
>   the current operator of newly-created entities, an instance of VertexOperator

entityTitleWidth
>   the entity title width multiplier, a value between 1.0 and 8.0

entityTitleFont
>   the document-wide entity title font, an instance of a FontSpec. This FontSpec can have a size of FontSpec.AUTOSIZE. Can be assigned to None to restore the default setting

entityClassFont
> the document-wide entity class name font, an instance of a FontSpec. Can be assigned to None to restore the default setting

groupTitleFont
> the document-wide group title font, an instance of a FontSpec. Can be assigned to None to restore the default setting

defaultAnnotationFont
> the document-wide default annotation font, an instance of a FontSpec. Can be assigned to None to restore the default setting

documentPath
> the path to the file from which the document was loaded, or None if the document has never been saved (read only)

hoistedGroup
> the currently hoisted group, a instance of Group, or None

hasSelection
> True if any elements in the graph are selected, otherwise False (read only)

selection
> the current selection in the graph as a list of GraphElement instances

all
> the entire graph as a list of GraphElement instances

orderedVertices
> returns all vertices as a list, earliest predecessor first, in acyclic order; i.e., ignoring back edges

reverseOrderedVertices
> returns all vertices as a list, latest successor first, in acyclic order; i.e., ignoring back edges. The reverseOrderedVertices list is not necessarily the reverse of the orderedVertices list because of how groups and unconnected entities are handled

domains
> a list of all the domains in the document (read only)

customSymbols
> a list of all the custom Symbols in the document (read only)

pageSize
> the currently calculated page size as a tuple of (width, height) when printing, in points (read only)

exportPath
> the directory that should be displayed in a file dialog involved in an export operation for this document, defaults to the user's home directory

imageExportAttributes `API 1.7`
> a dictionary with the current image export attributes (see Diagram Import Types in Application class)

chartFrame `API 1.7`
> the frame of the chart table as a tuple (*x, y, width, height*) (read only)

chartCornerFrame `API 1.7`
> the frame of the "corner" of chart table where the headers reside as a tuple (*x, y, width, height*) (read only)

chartRowFrame `API 1.7`
> the frame of the row header of chart table as a tuple (*x, y, width, height*) (read only)

chartColumnFrame `API 1.7`
> the frame of the columns of data in the chart table as a tuple (*x, y, width, height*) (read only)

chartColumnLines `API 1.7`
> a tuple containing the x-position of all vertical lines in chart table (read only)

chartHeaderTitles `API 1.7`
> a dictionary of the strings of the header labels (see Chart Part Types in Application class) (read only)

chartHeaderRects `API 1.7`
> a dictionary of the frames of the header labels as tuples (*x, y, width, height*) (see Chart Part Types in Application class) (read only)

zoomFraction `API 1.7`
> the current zoom value for the canvas. Values greater than 1.0 indicate the canvas is zoomed-in, values less than 1.0 indicate

the canvas is zoomed-out

modifyAttribute(list, name, value)

modify a particular built-in attribute *name* (a string) to *value* for every instance of GraphElem in *list*, throwing an exception if any instance does not support the given *name*. The following attributes can be set with this method:

```
weight
weightVisible
annotationVisible
operator
entityClass
symbol
color
confidence
completion
parent
collapsed
deepCollapsed*
startDate
finishDate
endDate**
effort
resource
```

The attributes `title` and `annotation` cannot be set via this method.

A bug existed that caused the selection to always be used for *list* when modifying project management attributes. This was fixed in Flying Logic 2.2.6.

*This represents a pseudo-attribute that can deep collapse a group

The attribute **endDate is deprecated; use **finishDate** instead.

modifyUserAttribute(list, name, value)

modify a particular user defined attribute *name* to *value* for every instance of GraphElem in *list*

hoistGroupFromSelection( )

hoists the first group found in the current selection. If there are more than one group in the selection, it is ambiguous which will be hoisted

hoistGroupToParent( )
> if a group is hoisted, its parent group is hoisted instead, otherwise does nothing

isSelected(elem)
> returns True if the GraphElem *elem* is selected, otherwise False

selectAll( )
> selects every element in the graph

clearSelection( )
> deselects every element in the graph

addEntity( )
> adds a new entity to the graph with class *selectedEntityClass*. If only one entity is currently selected, the new entity is connected to that selected entity per the setting of *addEntityAsSuccessor*. Returns a list of new elements, Entity instance first

addEntity(entityclass)
> adds a new entity to the graph with class *entityclass*. If only one entity is currently selected, the new entity is connected to that selected entity per the setting of *addEntityAsSuccessor*. Returns a list of new elements, Entity instance first

addEntityToTarget(vertexElem)
> adds a new entity to the graph with class *selectedEntityClass*. The newly created entity is connected to the given *vertexElem* per the setting of *addEntityAsSuccessor*. If *vertexElem* is None, does not connect the new entity to any element. Returns a list of new elements, Entity instance first

addEntityToTarget(entityclass, vertexElem)
> adds a new entity to the graph with class *entityclass*. The newly created entity is connected to the given *vertexElem* per the setting of *addEntityAsSuccessor*. If *vertexElem* is None, does not connect the new entity to any element. Returns a list of new elements, Entity instance first

insertEntity( )
> inserts a new entity to the graph with class *selectedEntityClass*, but only if a single edge is selected, otherwise an exception is thrown. Returns a list of new elements, Entity instance first

insertEntity(entityclass)
    inserts a new entity to the graph with class *entityclass*, but only if a single edge is selected, otherwise an exception is thrown. Returns a list of new elements, Entity instance first

insertEntityOnEdge(edge)
    inserts a new entity to the graph with class *selectedEntityClass* on the given *edge*. Throws an exception if *edge* is None. Returns a list of new elements, Entity instance first

insertEntityOnEdge(entityclass, edge)
    inserts a new entity to the graph with class *entityclass* on the given edge. Throws an exception if *edge* is None. Returns a list of new elements, Entity instance first

getDomainByName(name)
    return the Domain instance with the given *name* or None

getEntityClassByName(name_or_tuple)
    return the EntityClass instance based on one of two matching criteria: if *name_or_tuple* is a string, then the parameter is the name of an entity class to find (preference is given to a custom entity class if the are duplicate names); otherwise, if *name_or_tuple* is a tuple, then the parameter must be the tuple (domain_name, entity_class_name) identifying an entity class (see also the Domain class method *getEntityClassByName*)

Examples:

```
entityclass = document.getEntityClassByName('Goal')
entityclass = document.getEntityClassByName( \ ('Prerequisite
Tree, 'Milestone' ) )
```

print( )
    prints a document after displaying the print preferences dialog

print(ask)
    prints a document. Displays the print preferences dialog if *ask* is True

focusCanvas( )
    changes the current keyboard focus to the graph canvas

cut( )
    performs a cut operation on the selected elements in the graph

copy( )
    performs a copy operation on the selected elements in the graph

paste( )
    pastes the graph elements from the last, still active, copy operation to the document

deleteSelection(recurse)
    deletes the currently selected graph elements. If recurse is True, also deletes nested elements in selected groups

newDomain(name)
    returns a new Domain instance with the given *name*

deleteDomain(domain)
    deletes the given *domain*, automatically changing the class of any entity in the graph to defaultEntityClass if that entity's class was part of *domain*

deleteEntityClass (entityclass)
    deletes the given entity class, automatically changing the class of any entity in the graph to defaultEntityClass if that entity's class was *entityclass*. This is same operation as:

```
entityclass.getDomain().\ deleteEntityClass(entityclass)
```

newGroup( )
    creates a new group containing all currently selected elements and returns a list of new elements, Group instance first

newGroup(children)
    creates a new group containing all elements in children, which must be a list, and returns a list of new elements, Group instance first

newSymbol(path, rect)
    creates a new symbol from the file at *path* clipped to the *rect* tuple. If *path* is None, the user is asked to select a file. If *path* or *rect* is None, the Image Viewer dialog is shown. The *rect* tuple is (left, top, width, height)

newSymbolFromObject(obj, rect)
    creates a new symbol from *obj*, where *obj* can either be an instance of java.awt.Image, javax.swing.ImageIcon or a string of a SVG XML document, clipped to the *rect* tuple. The *rect* tuple is

(left, top, width, height)

deleteSymbol(symbol)
deletes the symbol, fixing-up all entity classes and entities as needed

isCustomSymbol(symbol)
returns True if *symbol* is custom (not built-in)

connect(fromElem, toElem)
connects an edge from the *fromElem* to the *toElem*, where the elements must be an entity, junctor or edge. Returns a list of new elements

reconnect(edge, part, element)
reconnects one end of *edge* to a new element, where *part* indicates which end (see Edge Part Type in Application class). Returns a list of new elements

saveDocument( )
saves the document, asking the user to select a file only if the document has never been saved

saveDocumentAs(path)
saves the document to the file at *path*, creating the file if necessary. If *path* is None, the user is asked to select a file

saveDocumentAsTemplate(path)
saves the document as a template to the file at *path*, creating the file if necessary. If *path* is None, the user is asked to select a file

exportDocument(kind, path, params)
exports a document to a file of *kind* at *path* with settings in the *params* dictionary. If *path* is None, the user is asked to select a file. The *kind* is one of the Export Types in the Application class. The keys and values in *params* are export kind dependent (see Image Export, OPML Export, and XSLT Import/Export Types for possible keys and values)

importDocument(kind, path, params)
imports a document from a *file* of kind at *path* with settings in the *params* dictionary. If *path* is None, the user is asked to select a file. The *kind* is one of the Import Types in the Application class. The keys and values in *params* are export kind dependent

(only XSLT import used params, XSLT Import/Export Types for possible keys and values). Returns either a new Document instance, if the import creates a new document, or this Document if not

createResource( )
creates and returns a new Resource instance with default values. If you change the values you must call updateResource

createResource(name, abbreviation, utilization, calendar)
creates and returns a new Resource instance with the given values. The name and utilization parameters must not be None, but abbreviation and calendar can. If a resource with name already exists, the value "Copy of" prepended to name is used. For a default utilization supply the value 1.0. If calendar is None, the standard calendar is assigned to the new resource

copyResource(resource)
returns a copy of the given resource with name changed by prepending Copy of"

updateResource(resource)
updates a Resource instance. Changes to resource instance do not take full effect until this method is called

removeResource(resource)
removes the given resource from the document

resourceByName(name)
return the Resource instance with the given name or None if these is no such resource

createCalendar( )
creates and returns a new Workweek instance with default values. If you change the values you must call updateCalendar

createCalendar(name, workdays, workhours)
creates and returns a new Workweek instance with the given values. The name parameter must not be None, If a calendar with name already exists, the value "Copy of" prepended to name is used. The workdays parameter must be an combination of day of week values and cannot be zero. The workhours parameter must be between 1.0 and 23.0

copyCalendar(calendar)
> returns a copy of the given calendar with name changed by prepending Copy of"

updateCalendar(calendar)
> updates a Workweek instance. Changes to calendar instance do not take full effect until this method is called

removeCalendar(calendar)
> removes the given calendar from the document

calendarByName(name)
> return the Workweek instance with the given name or None if these is no such calendar

resetStartTime( )
> resets the project start time to the start of the work day. Only applies when finish-to-start scheduling is set

resetFinishTime( )
> resets the project finish time to the end of the work day. Only applies when start-to-finish scheduling is set

closeDocument(ask)
> closes the document, asking the user for permission is *ask* is True and the document has been modified

find(match, options)
> finds all graph elements that correspond to *match* with the given options (see Find Types in Application class)

selectSuccessors( )
> selects the successor of all current selected entities and junctors, including any edge in-between

selectPredecessors( )
> selects the predecessor of all current selected entities and junctors, including any edge in-between

selectEdgeHeadEntity( )
> selects the head entity of all current selected edges, including any edge or junctor in-between

selectEdgeTailEntity( )
> selects the head entity of all current selected edges, including

any edge or junctor in-between

reverseSelectedEdges( )
the head and tail elements of edge selected edge are swapped

swapSelectedElements( )
if the two selected elements are of the same type, swaps them

swapSelectedForwardAndBackEdges( )
swaps the selected forward and back edges

redactSelection( )
redact the selected elements

redactAll( )
redact all elements

importDomain(path)
imports the domain from the file at *path*

saveDomainsAsDefaults( )
saves the current custom domains as the defaults for future new documents

eraseProjectManagement( )
erases all project management information from the document, the same affect as:

`document.startData = None`

isSymbolInUse(symbol)
return True if a *symbol* is being used by an EntityClass or Entity

getSymbolByName(name)
returns the a Symbol instance matching the given *name*. Symbol names are a generator name and an ID code separated by a colon. For example, the blue pentagon symbol is "com.arciem.symbol.FlowchartSymbolGenerator:pentagon". If name is "inherit" or "none", the predefined special symbol values Application. INHERIT_SYMBOL and Application.NO_SYMBOL,respectfully. A table with the names of the builtin symbols can be found in .

undo( )
perform an undo operation if possible

redo( )

  perform a redo operation if possible

beginCoalesceUndo(name)

  causes all changes to a document in a script to be coalesced into one undo record under the given *name*. This is done internally for each document accessed by a script, but can still be called to change the undo record's name. The method endCoalesceUndo is called internally when the script terminates

endCoalesceUndo( )

  closes and appends the current undo record to the undo stack, but only if there is anything to undo. A new undo record is automatically started, but it's name can be changed by calling beginCoalesceUndo

operate(operator, flags)

  operate on each element in a graph in acyclic order using *operator*, an instance of a class derived from GraphOperator, with the types and order of elements determined by *flags* (see Operate Types in Application class)

formatDate(date) `API 1.7`

  returns a string representing the given date as formatted for a start or finish date

calcFont(spec) `API 1.7`

  calculates the ascent, descent and uppercaseHeight fields of the given FontSpec spec derived from the document environment

getStringBounds(string, spec) `API 1.7`

  calculates the bounds of the given string rendered with the given FontSpec spec. Returns a tuple (x, y, width, height)

truncateWithEllipsis(string, spec, width) `API 1.7`

  if the given string rendered the given FontSpec spec does not fit in width, returns the string truncated and an ellipsis added. Return the original string is it would fit

findStaticFont(name) `API 1.7`

  returns a FontSpec for the given UI usage name, or None is no such font

inHoist(elem) `API 1.7`

  returns True if the given GraphElem elem is in the current hoist;

i.e., the current hoisted group is an ancestor of elem. Otherwise, returns False

Application

The Application class represents those features of Flying Logic that are independent of any particular document. Such features include application preferences, document loading and importing, version information, etc. In addition the Application class contains constants used in method calls in the Application class and others.

There is only one instance (singleton) of the Application class. This instance can be accessed by the global variable Application.

There a number of types (orientation type, bias type, etc.) accessible from the Application instance. These can be found listed after the variables and methods section of the class.

version
> the version of Flying Logic as a string (read only)

apiVersion
> the version of Flying Logic scripting API as a string (read only). This string is "1.7" in Flying Logic Pro 3.0.6 **API 1.7**

edition
> the edition of Flying Logic as a string, either "Pro" or "Reader" (read only)

language
> the user interface language of Flying Logic as a two-letter ISO 639 code possibly with a hyphen and country code appended to the end (read only)

vertexOperators
> a list of all available vertex operators (read only)

defaultOrientation
> the default orientation of new documents preference value (see possible values under orientation types)

defaultBias
> the default bias of new documents preference value (see possible values under bias types)

animationSpeed
> the animation speed preference value, between 0.0 and 1.0

adaptiveSpeed
the adaptive speed preference value, between 0.0 and 1.0

adaptiveAnimation
the adaptive animation state preference value, either True or False

animationStyle
the animation style preference value (see possible values under animation style types)

edgeColors
the edge colors preference value (see possible values under edge colors types)

spinnerDisplay
the spinner display preference value (see possible values under spinner display types)

undoLevels
the maximum number of undo levels to retain preference value, an integer

autoBackupOnSave
the auto-backup on save preference value, either True or False

checkForUpdates
the check for updates preference value, either True or False

useProxyServer
the use proxy server preference value, either True or False

proxyServer
the proxy server preference value, either a domain name or an IP address as a string

proxyPort
the port of the proxy server preference value, an integer

maxRecentDocuments
the maximum number of documents in Open Recent menu preference value, an integer

maxrecentScripts

the maximum number of scripts listed in the Run Script sub-menu, as an integer

maxrecentImports
the maximum number of scripts listed in the Import Via Script sub-menu, as an integer

maxrecentExports
the maximum number of scripts listed in the Run Export Via Script sub-menu, as an integer

autoEditNewEntityTitles
the auto edit entity titles preference value, either True or False

canDisableControlAltShortcuts
the can disable ctrl-alt menu shortcuts preference value, either True or False. Always returns False under Mac and Linux

importPath
the directory that should be displayed in a file dialog involved in an import operation for the application, defaults to the user's home directory

lastAskDirectory
the directory the user chose during the last call to askForFile, a string (read only)

entityFilter
an instance of GraphElemFilter that only matches entities (read only)

junctorFilter
an instance of GraphElemFilter that only matches junctors (read only)

groupFilter
an instance of GraphElemFilter that only matches groups (read only)

edgeFilter
an instance of GraphElemFilter that only matches edges (read only)

startFilter
an instance of GraphElemFilter that only matches entities with

no predecessors (read only)

endFilter

an instance of GraphElemFilter that only matches entities with no successors (read only)

defaultDocumentPath

the default location for storing documents as a string (read only)

defaultWorkHours

the default work hours for a newly created Workweek instance including the standard calendar when project management is first enabled for a document

openDocumentOption

the preferences for whether new documents should be opened in a window, tab, or the user always queried for the choice (when possible).

newDocument()

returns a Document instance representing a newly created Flying logic document

openDocument(path)

returns a Document instance representing the Flying Logic document opened from the given *path*. If the document is already opened, this method just returns the existing Document instance

findDocument(path)

returns a Document instance representing the open Flying Logic document opened from the given *path*, otherwise None is the document is not open

importDocument(path, params)

returns a Document instance representing a Flying Logic document created by importing the file with given *path* (or by asking the user for a file if path is None) using an XSLT file to transform the file into a Flying Logic document based on options in the *params* dictionary (see possible options under XSLT Import/Export Types)

showQuickCapture()

displays the Quick Capture dialog

exporterLabel(exportType)
> returns the label for the export menu item matching the given *exportType*

vertexOperatorByName(name)
> returns the VertexOperator with the given user interface *name*; i.e., "Fuzzy And"

filterGraphElemList(list, filter)
> given a *list* of graph elements returns a new list that has been filtered by the given GraphElemFilter *filter*

alert(message)
> displays a simple alert dialog with a string *message*

askForString(message, defaultValue)
> displays a simple dialog with *message* requesting the user enter a value, which is returned as a string. If the user selects Cancel, the *defaultValue* is returned instead. The value None is an acceptable *defaultValue*

askForInteger(message, defaultValue)
> displays a simple dialog with *message* requesting the user enter a value, which is returned as an integer. If the user selects Cancel, the *defaultValue* is returned instead

askForDouble(message, defaultValue)
> displays a simple dialog with *message* requesting the user enter a value, which is returned as a float (which is equivalent to a Java double). If the user selects Cancel, the *defaultValue* is returned instead

request(message, labels)
> displays a request dialog titled "Request" to the user to answer a question by making a selection among a set of buttons with *labels* (a tuple). Returns an integer matching the index of the label in the tuple. Note: the *message* can be a string or a Java Component object, allowing for the creation of a more complicated dialog

request(title, message, labels)
> displays a request dialog with *title* to the user to answer a question by making a selection among a set of buttons with *labels* (a tuple). Returns an integer matching the index of the label in the

tuple. Note: the *message* can be a string or a Java Component object, allowing for the creation of a more complicated dialog

askForFile(defaultDirectory, save)

displays a file dialog requesting the user select a file. The dialog is a save file dialog if save is True, else it's an open file dialog. The dialog initially shows the files in *defaultDirectory* or the default user directory if None. Returns the path to the selected file as a string or None if the user cancelled

appendClassPath(pathList)

appends the paths in pathList to the Java classpath for the script. The values in the list should be either directories containing Java class files or paths to jar files, as strings

askForFile(save)

displays a file dialog just like the above method, but always initially shows the files in the default user directory

createCompatibleDriver(driver)

returns a "compatible" instance of java.sql.Driver that acts as a shim class to another instance of java.sql.Driver created from a JDBC library. This is a workaround for a "feature" of Java where Driver instances can only be used if created by the application ClassLoader, which is not true of scripts running in Flying Logic

```
# You should have previously added the path to the
    MySQL JDBC jar via Application.appendClassPath
    method
from com.mysql.jdbc import Driver
from java.sql import DriverManager

# Need to create shim Driver
shimDriver = Application.createCompatibleDriver( Driver() )
DriverManager.registerDriver(shimDriver);
conn = DriverManager.getConnection("jdbc:mysql://someserver/
somedb", "someuser", "somepassword")
```

createSQLConnection(url, username, password, driverClassName)

returns a Python SQL connection object. This method is a replacement for the the connect method in Jython's zxJDBC package.

```
# You should have previously added the path to the MySQL JDBC
jar via Application.appendClassPath method
url = "jdbc:mysql://someserver/somedb"
```

```
username = "someuser"
password = "somepassword"
driver = "com.mysql.jdbc.Driver"

# obtain a connection using the with-statment
#with zxJDBC.connect(jdbc_url, username, password, driver) as
conn:
with Application.createSQLConnection(url, username, password,
driver) as conn:
    with conn:
        with conn.cursor() as c:
         # execute SQL commands
```

Orientation Types

```
ORIENTATION_LEFT_TO_RIGHT
ORIENTATION_RIGHT_TO_LEFT
ORIENTATION_TOP_TO_BOTTOM
ORIENTATION_BOTTOM_TO_TOP
ORIENTATION_INNER_TO_OUTER
ORIENTATION_OUTER_TO_INNER
```

Example:

```
Application.defaultOrientation = Application.ORIENTATION_LEFT_
TO_RIGHT
```

Bias Types

```
BIAS_START
BIAS_END
```

Example:

```
Application.defaultBias = Application.BIAS_START
```

Animation Style Types

```
ANIMATION_FIXED_FRAME_RATE
ANIMATION_FIXED_TIME
```

Example:

```
Application.animationStyle = Application.ANIMATION_FIXED_FRAME_
RATE
```

Edge Colors Types

```
EDGE_RED_GRAY_BLACK
EDGE_RED_YELLOW_BLACK
EDGE_RED_YELLOW_GREEN
```

Example:

```
Application.edgeColors = Application.\
EDGE_RED_GRAY_BLACK
```

Spinner Display Types

```
SPINNER_DISPLAY_NONE
SPINNER_DISPLAY_NUMERIC
SPINNER_DISPLAY_SYMBOL
```

Example:

```
Application.spinnerDisplay = Application.\
SPINNER_DISPLAY_SYMBOL
```

Export Types

```
EXPORT_DIAGRAM_PDF
EXPORT_DIAGRAM_JPEG
EXPORT_DIAGRAM_PNG
EXPORT_DIAGRAM_DOT
EXPORT_ANNOTATIONS_PDF
EXPORT_ANNOTATIONS_TEXT
EXPORT_OUTLINE_OPML
EXPORT_DIAGRAM_PROJECT_XML
EXPORT_DIAGRAM_PROJECT_MPX
EXPORT_DIAGRAM_XSLT
```

Example:

```
document.exportDocument( Application.EXPORT_DIAGRAM_PDF, None,
() )
```

Import Types

```
IMPORT_DIAGRAM_CSV
IMPORT_DIAGRAM_XSLT
```

Example:

```
document.importDocument( IMPORT_DIAGRAM_CSV,None, () )
```

Image Export Types

```
IMAGE_EXPORT_WIDTH
IMAGE_EXPORT_HEIGHT
IMAGE_EXPORT_RESOLUTION
IMAGE_EXPORT_SHOW_SELECTION
IMAGE_EXPORT_SAVE_INK
```

These types are used as keys in a dictionary, the value being the setting for the key. If a type appears in the dictionary, it overrides the current setting.

Example:

```
params = ( Application.IMAGE_EXPORT_SHOW_SELECTION : False )
document.exportDocument(Application.EXPORT_DIAGRAM_JPEG, params)
```

Weekday Type

```
SUNDAY_MASK
MONDAY_MASK
TUESDAY_MASK
WEDNESDAY_MASK
THURSDAY_MASK
FRIDAY_MASK
SATURDAY_MASK
DEFAULT_WORKDAYS
```

These are the same constants found in the Workweek class and duplicated here for convenience. See Workweek class for details.

OPML Export Types

```
OPML_EXPORT_FORWARD
OPML_EXPORT_UNICODE
OPML_EXPORT_INCLUDE_EQUATION
```

These types are used as keys in a dictionary, the value being the setting for the key. If a type appears in the dictionary, it overrides the current setting.

Example:

```
params = ( Application.EXPORT_OUTLINE_OPML: False )
document.exportDocument(Application.EXPORT_DIAGRAM_JPEG, params)
```

XSLT Import/Export Types

```
XSLT_ASK
XSLT_FILE
XSLT_STRING
XSLT_INCLUDE_FRAMES
XSLT_INCLUDE_EDGE_SPLINES
```

These types are used as keys in a dictionary, the value being the setting for the key.

The first three values are mutually exclusive and indicate the source of the XSLT file: either ask the user for locate the file (value should be True), use the given file (the value is the path to the file as a string), or use the already loaded/embedded string (XLST file as a string).

The last two are only used when exporting, and indicate whether the Flying Logic document to be transformed should include graph element frames and edge splines.

Example:

```
params = (
    Application.XSLT_ASK : True,
    Application.XSLT_INCLUDE_FRAMES : True
)
document.exportDocument(Application.EXPORT_DIAGRAM_XSLT, params)
```

Find Types

```
FIND_CASE_SENSITIVE
```

```
FIND_WHOLE_WORDS_ONLY
FIND_SEARCH_LABELS
FIND_SEARCH_ANNOTATIONS
FIND_SEARCH_UDA_NAMES
FIND_SEARCH_UDA_VALUES
FIND_SEARCH_RESOURCES
FIND_SELECT_COLLAPSED_GROUPS
```

These types are used as keys in a dictionary, the value being the setting for the key. If a type appears in the dictionary, it overrides the current setting.

Example:

```
params = ( Application.FIND_SEARCH_ANNOTATIONS:True )
document.find('communicate', params)
```

Operate Types

```
OPERATE_ENTITY
OPERATE_JUNCTOR
OPERATE_EDGE
OPERATE_REVERSE
OPERATE_NON_EDGE
OPERATE_ALL
```

The first four types above are bit fields. The first three limit which graph element types are operated upon, while OPERATE_REVERSE indicated the elements should be iterated through end to start. OPERATE_NON_EDGE is the same as OPERATE_ENTITY | OPERATE_JUNCTOR, and OPERATE_ALL should be obvious.

Example:

```
mask = Application.OPERATE_ENTITY | Application.OPERATE_REVERSE
document.operate(myGraphOperator, mask)
```

Symbol Name Types

```
BITMAP_PREFIX
SVG_PREFIX
SYMBOL_SEPARATOR
```

BITMAP_PREFIX and SVG_PREFIX are the generator names for bitmap images and SVG drawing custom symbols. SYMBOL_

SEPARATOR is the a colon.

See for a list of the built-in symbol name constants.

Special Symbol Types

```
INHERIT_SYMBOL
NO_SYMBOL
```

Special predefined Symbol instances. INHERIT_SYMBOL will set an entity's symbol back to the default for its entity class. NO_SYMBOL will set an entity's symbol to none, overriding any possible default symbol for its entity class. For groups both IN-HERIT_SYMBOL and NO_SYMBOL clear the group symbol.

Text Editor Types

```
BOLD
ITALIC
UNDERLINED
STRIKETHROUGH
LINK
FONT_SIZE
FONT_FAMILY
FONT_COLOR
```

These types are used as keys in a dictionary, the value being the attribute for the key.

Example:

```
editor.changeSelectionAttributes( ( Application.BOLD: True ) )
```

Edge Part Types

```
EDGE_HEAD
EDGE_TAIL
```

Which end of an edge to reconnect.

Example:

```
document.reconnect( edge, Application.EDGE_TAIL, entity )
```

Shape Part Types `API 1.7`

```
PART_WEIGHT
PART_ANNOTATION
PART_HOIST_CAP
PART_CONFIDENCE
PART_COMPLETION
PART_ENTITYID
PART_TITLE
PART_CLASS
PART_CLASS_BKGD
PART_START_DATE
PART_FINISH_DATE
PART_RESOURCES
PART_DISCLOSURE
PART_SYMBOL
PART_ANNOTATION_NUMBER
PART_ANNOTATION_TEXT
PART_COMPLETION_BAR
PART_EDGE_INFO
```

Parts of an element's shape.

Chart Part Types `API 1.7`

```
PART_CHART_ROW
PART_CHART_INDEX
PART_CHART_ANNOTATION
PART_CHART_DISCLOSURE
PART_CHART_SYMBOL
PART_CHART_TITLE
PART_CHART_CLASS
PART_CHART_CLASS_COLOR
PART_CHART_CONFIDENCE
PART_CHART_ENTITYID
PART_CHART_START_DATE
PART_CHART_FINISH_DATE
PART_CHART_COMPLETION
PART_CHART_RESOURCES
PART_CHART_EFFORT
```

Parts of the chart table.

Other Font Types

AUTOSIZE

A special "font size" that means "set the size as the application sees fit."

Example:

`document.titleFontSize = Application.AUTOSIZE`

File Types

**FILE_SEPARATOR
DOCUMENT_EXTENSION
TEMPLATE_EXTENSION
DOMAIN_EXTENSION**

These are various file-lated constants: the system-dependent file path separator, the Flying Logic document file extension, the Flying Logic template file extension, and the Flying Logic domain file extension.

Schedule Types

**SCHEDULE_FROM_START_DATE
SCHEDULE_FROM_FINISH_DATE
~~SCHEDULE_FROM_END_DATE~~** *deprecated*

Direction that project should be scheduled.

Open Document Option Types

**OPEN_DOCUMENT_IN_WINDOW
OPEN_DOCUMENT_IN_TAB
OPEN_DOCUMENT_ASK**

valid values for the openDocumentOption preference

Canvas View Types

**GRAPH_VIEW
CHART_VIEW**

Canvas view kinds.

Resource Assignment Types

```
RESOURCE_FIXED_EFFORT
RESOURCE_FIXED_DURATION
RESOURCE_FIXED_EFFORT_DURATION
```

If two or more resources are assigned to a task, this value determines how those resources are applied. With RESOURCE_FIXED_EFFORT the effort is considered total work hours to be distributed between resources. With RESOURCE_FIXED_DURATION, the effort acts as a fixed duration and resources all work that number of hours on the task. Finally, RESOURCE_FIXED_EFFORT_DURATION results in each resource only working part-time on the task.

Edge Spline Types `API 1.7`

```
PART_HEAD
PART_MIDDLE
PART_TAIL
```

Types that appear in a spline dictionary.

Diagram Import Types `API 1.7`

```
DIAGRAM_IMPORT_NEW_DOCUMENT
DIAGRAM_IMPORT_PM_DOMAIN
DIAGRAM_INCLUDE_TOP_GROUP
```

These types are used as keys in a dictionary, the value being the setting for the key. If a type appears in the dictionary, it overrides the current setting. All the current settings are booleans.

Static Font Types `API 1.7`

```
FONT_SPINNER_LARGE
FONT_SPINNER_SMALL
FONT_SPINNER_UNDEFINED
FONT_ANNOTATION_ICON
FONT_ANNOTATION_NUMBER
FONT_DATE
FONT_DATE_BOLD
```

```
FONT_ROW
FONT_ROW_BOLD
FONT_CAP_ENTITY_ID
FONT_JUNCTOR_LARGE
FONT_JUNCTOR_SMALL
FONT_JUNCTOR_CHART
FONT_CHART_CALENDAR
```

Symbolic names of various fonts used to display shapes. Used in Document's findStaticFont method.

Symbol Scale and Size Types `API 1.7`

```
GRAPH_SYMBOL_SCALE
CHART_SYMBOL_SCALE
```

Scale factors when calling generateSvg method of Symbol class.

```
CHART_SYMBOL_SIZE
```
`API 1.7`

Can be used as maxSize when calling generateSvg method of Symbol class for chart-like output. Graph-like output should set maxSize to None indicating an unconstrained size.

GraphElem

The GraphElem is the base class for all representations of graph elements: entities, junctors, groups and edges. This class has variables and methods common to all elements.

eid
> unique integer assigned to each GraphElem (read-only)

isEntity
> returns False (see Entity, read only)

isJunctor
> returns False (see Junctor, read only)

isGroup
> returns False (see Group, read only)

isEdge
> returns False (see Edge, read only)

canHaveParent
> returns False, (see VertexElem, read only)

hasParent
> returns True if has parent (read only)

frame
> the element's bounds as a tuple (left, top, width, height), or (0, 0, 0, 0) if the element is hidden in a collapsed group (read only)

annotation
> the annotation (note) of the element, an HTML document as a string, or None if the element has no note. If set to plain text, the font of the annotation defaults to Monospaced/12

isHiddenInCollapse
> returns True if the element is in a collapsed group, otherwise False

hasAnnotation
> returns True if the element has an annotation, otherwise False

annotationNumber
> the annotation (note) number of the element, an integer, or zero if the element has no note (read only)

displayAnnotationNumber `API 1.7`

> the annotation (note) number of the element if note numbers are displayed as an integer, or zero if the element has no note or note numbers are not displayed (read only)

partFrames `API 1.7`

> a dictionary of tuples (*x, y, width, height*). The keys are Shape Part Types (see Application class) for visual elements that vary by GraphElem type and state (read only)

plainAnnotation

> the annotation (note) of the element as plain text, or None if the element has no note (read only)

annotationEditor

> returns a TextEditor instance which allows for modification of the annotation, or None if the element has no note (variable is read only, the TextEditor itself can be modified)

user

> a dictionary containing user defined attributes (variable is read only, the dictionary itself can be modified)

> Examples:

```
previousFacility = elem.user['facility']
elem.user['facility'] = 'Los Angeles'
del elem.user['careless']
```

Edge

Edge is derived class of GraphElem and represents an edge in the graph.

isEdge
> returns True (read only)

weight
> the edge weight, a float value normally between -1.0 and 1.0

weightVisible
> True if the edge weight is visible independently of the document's edgeWeightsVisible setting

annotationVisible
> True if the edge annotation is visible independently of the document's edgeAnnotationsVisible setting

source
> the source element of the edge; i.e., the element at the tail-end of the edge (read only)

target
> the target element of the edge; i.e., the element at the head-end of the edge (read only)

isBackEdge
> true if the edge is a back edge (read only)

~~**splines**~~ `API 1.7`
> deprecated. Use spliesDictionary instead

splinesDictionary `API 1.7`
> the edge's splines as a dictionary or None if the edge is hidden in a collapsed group. The keys are the The dictionary values are tuples of tuples of the spline control points (read only)

arrowheadVertices `API 1.7`
> the edge's arrowhead as a tuple of three tuples (*x, y*) or None if the edge is hidden in a collapsed group (read only)

color `API 1.7`
> the edge's color in the canvas

noteLines `API 1.7`
> a list of LayoutLines providing position information for the an-

notation (read only)

isWeightVisible() `API 1.7`
> True if the edge weight is visible either because weightVisible is True or the document's edgeWeightsVisible setting is True, otherwise False (read only)

isAnnotationVisible() `API 1.7`
> True if the edge annotation is visible either because annotationVisible is True or the document's edgeAnnotationsVisible setting is True, otherwise False (read only)

equals(object)
> returns True if *object* represents the same edge as self

VertexElem

VertexElem is derived class of GraphElem and is a base class representing an entity, junctor and group.

canHaveParent
> returns True (read only)

parent
> the parent Group of the element or None if the element is at the top of the hierarchy

hasInEdges
> True if the element has any predecessors, implying it has at least one in-edge. Ignores back edges during a call to the operate method of Document (read only)

hasOutEdges
> True if the element has any successors, implying it has at least one out-edge. Ignores back edges during a call to the operate method of Document (read only)

inEdges
> a list of the element's in-edges. Does not include back edges during a call to the operate method of Document (read only)

outEdges
> a list of the element's out-edges. Does not include back edges during a call to the operate method of Document (read only)

index
> the index of the element in chart view or zero if canvas not in chart view (read only)

Entity

Entity is derived class of VertexElem and represents an entity.

isEntity
> returns True (read only)

title
> the title, as a string

confidence
> the confidence, as a value between 0.0 and 1.0

canDrive
> True if this entity can drive confidence; i.e., it has no predecessors ignoring back edges (read only)

completion
> the completion, as a value between 0.0 and 1.0. 0.0 if project management not enabled

effortHours
> the effort in hours, as a positive floating-point value or zero for a milestone. Zero if project management not enabled.

~~effort~~
> *this field is deprecated* — use **effortHours** instead. The effort in days where one day is the document default work hours, as a positive value or zero for a milestone. Zero if project management not enabled

startDate
> the start date, as a Date. None if project management not enabled

endDate
> the end date, as a Date. None if project management not enabled

resources
> a list of resources assigned to a task. The list can be empty (read only)

resourceAssignment
> the current setting for how resources are assigned to a task. See *Resource Assignment Type* in *Application* class for more details

symbol

the directly set symbol for this entity, as a Symbol, otherwise None

inheritedSymbol

the symbol for this entity, either the directly set Symbol or the one inherited from its EntityClass. None if both values are None (read only)

entityClass

the class, as an EntityClass

entityID

the element's entity ID, as an integer. Can be zero if entityID are not currently visible (read only)

equals(object)

returns True if *object* represents the same entity as self

addResource(resource)

assigns the given resource to the task

removeResource(resource)

removes the given resource assigned to the task

chartRowEffortString `API 1.7`

the string that would be displayed for the entity's effort in chart view or None is project management not enabled (read only)

preferredStartDate `API 1.7`

the preferred start date, as a Date. None if the entity has no preferred start date or project management not enabled

preferredFinishDate `API 1.7`

the preferred finish date, as a Date. None if the entity has no preferred finish date or project management not enabled

preferredDateError `API 1.7`

set to True if the entity exhibits a date error, otherwise False (read only)

resourceString

the string that w `API 1.7` ould be displayed for the entity's resources in graph view, or None if no resources assigned or project management not enabled (read only)

chartRowResourceString `API 1.7`

the string that would be displayed for the entity's resources in chart view, or None if no resources assigned or project management not enabled (read only)

startDateString `API 1.7`

the string that would be displayed for the entity's start date, or None if project management not enabled (read only)

finishDateString `API 1.7`

the string that would be displayed for the entity's finish date, or None if project management not enabled (read only)

titleFont `API 1.7`

the FontSpec used to render the entity title (read only)

titleLines `API 1.7`

a list of LayoutLines providing position information for the entity title (read only)

Junctor

Junctor is derived class of VertexElem and represents a junctor.

isJunctor
   returns True (read only)

operator
   the operator, as a VertexOperator

Group

Group is derived class of VertexElem and represents a group.

isGroup
    returns True (read only)

title
    the title, as a string

color
    the background color, as a Color

symbol
    the symbol for this group, as a Symbol, otherwise None

collapsed
    True if the group is collapsed

children
    the elements contained in the group, as a list of VertexElem (read only)

startDateString
    the string that would be displayed for the group's start date, or None if project management not enabled (read only)

finishDateString
    the string that would be displayed for the group's finish date, or None if project management not enabled (read only)

titleFont
    the FontSpec used to render the group title (read only)

titleLines
    a list of LayoutLines providing position information for the group title (read only)

deepCollapse( )
    performs a deep collapse on the group

deepExpand( )
    performs a deep expand on the group

Domain

The Domain class represents a domain, a named collection of entity classes, in a document.

name
　the domain's name, as a string

builtIn
　True if the domain is built-in and cannot be modified (read only)

hidden
　True if hidden

entityClasses
　entity classes in the domain, as a list of EntityClass

getEntityClassByName(name)
　returns the entity class in the domain with *name*, as an Entity-Class. Returns None if there is no such class

newEntityClass( )
　creates and returns a new entity class in the domain, as an EntityClass

deleteEntityClass(entityclass)
　deletes the given *entityclass* from the domain

export(path)
　exports the domain to a the file at *path*, asking the user to select a file if path in None

duplicate( )
　returns a duplicate of the domain with the name modified to include "copy," as a Domain

EntityClass

The EntityClass class represents an entity class.

name
>   the class name, as a string

builtIn
>   True if the domain is built-in and cannot be modified (read only)

color
>   the background color behind the class title, as a Color

weight
>   the default weight for new edges created along with entities of this class, as a value between -1.0 and 1.0

symbol
>   the symbol inherited by new entities of this class, as a Symbol

milestone
>   a boolean value, true if entities derived from this class should be a milestone (zero effort) task, otherwise false

resourceAssignment
>   the default resource assignment for entities derived from this class should be a milestone (zero effort) task, otherwise false

~~**effort**~~
>   this field is deprecated and can return different values than Flying Logic 2. Milestones have a value of 0.0, otherwise 28800.0 (8 hours in seconds)

showName
>   a boolean value indicating if the class name should be shown, defaults to True

domain
>   the parent domain of this class, as a Domain (read only)

nameColor `API 1.7`
>   the color that the name of the entity class would be rendered in an entity (read only)

duplicate( )
>   creates and returns a unique duplicate of this class with the

name modified to include "copy," as an EntityClass

Symbol

The Symbol class represents a symbol. Symbol instances are immutable.

symbolID
>the unique ID for this symbol, as a string (read only)

identifier
>a string suitable as a parameter to the Document method get-SymbolByName (read only)

equals(object)
>returns True if *object* represents the same symbol as self

generateSvg(document, x, y, scale, maxSize) `API 1.7`
>generate an SVG fragment that renders the symbol at the given position with given scale factor (see Symbol Scale and Size Types in Application class) and maxSize, a tuple of (width, height) and can be None

Color

The Color class represents an RGB color value. Color instances are immutable. The constructors are accessible from scripts.

r
> the red component, as a value between 0.0 and 1.0 (read only)

g
> the green component, as a value between 0.0 and 1.0 (read only)

b
> the blue component, as a value between 0.0 and 1.0 (read only)

a
> the alpha component, as a value between 0.0 and 1.0 (read only)

init(r, g, b)
> constructs a new Color instance with the given red, green and blue values (in the range 0.0 to 1.0) and an alpha of 1.0

init(r, g, b, a)
> constructs a new Color instance with the given red, green, blue and alpha values (in the range 0.0 to 1.0)

init(name)
> constructs a new Color instance derived from standard HTML color hex value or color *name*; e.g., "#FF0000", "white", "black", "orange", etc. (see "Online Resources" on page 13 for a link to a complete list.)

equals(object)
> returns True if *object* represents the same color as self within a set tolerance

Pre-defined colors

> These are class variables of Color. The red, green and blue components are given after each variable name. Note that some are not a match for the HTML color name equivalent; e.g., ORANGE is the not the same color as HTML "orange"

```
BLACK          0.00, 0.00, 0.00
WHITE          1.00, 1.00, 1.00
```

```
RED          1.00, 0.00, 0.00
ORANGE       1.00, 0.50, 0.00
YELLOW       1.00, 1.00, 1.00
GREEN        0.00, 1.00, 0.00
BLUE         0.00, 0.00, 1.00
VIOLET       1.00, 0.00, 0.50
CYAN         0.00, 1.00, 1.00
MAGENTA      1.00, 0.00, 1.00
GRAY         0.50, 0.50, 0.50
LIGHT_GRAY   0.75, 0.75, 0.75
DARK_GRAY    0.25, 0.25, 0.25
```

Date

The Date class represents a day of the year. Date instances are immutable. The constructors are accessible from scripts.

day
  the day, an integer from 1 to 31 (read only)

month
  the month, an integer from 0 to 11 (read only)

year
  the year, an integer (read only)

weekday
  the day of the week, an integer from 0 to 6 (read only)

init( )
  constructs a new Date instance for the current date

init(year, month, day)
  constructs a new Date instance for the given *year*, *month* and *day*

equals(object)
  returns True if *object* represents the same date as self

compareTo(date)
  returns 0 if the *date* represented by the argument is equal to the date represented by this Date, less than 0 if the date of this Date is before the date represented by the argument,and greater than 0 if the date of this Calendar is after the date represented by the argument

addDays(days)
  returns a new Date instance representing a date with the given number of *days* added to the date represented by this Date. The argument days can be negative

addDays(workdays, workweek)
  returns a new Date instance representing a date with the given number of *workdays* added to the date represented by this Date and properly considering the Workweek represented by the parameter *workweek*. The argument *workdays* can be negative.

daysFromSunday `API 1.7`
>   the day of the week as a difference from Sunday: 0 for Sunday, 1 for Monday, etc. (read only)

Workweek

The Workweek class represents the information about what the work schedule for project management calculations. The constructors are not accessible from scripts.

workdays
>   the days of the week that are work days, as a bit field (see Workday Type below)

events
>   the exceptions to the work schedule, as a list of CalendarEvent (read only)

init(workdays)
>   creates a new Workweek with the given *workdays* with no work schedule exceptions

init(workweek)
>   creates a new Workweek which is a deep copy of given Workweek instance

checkWorkDay(day)
>   returns True if the given day is a regular work day for this Workweek, where *day* is one of the Workday values below

checkWorkDay(date)
>   returns True if the given *date* (a Date instance) would be a work day for this Workweek with exceptions considered

addExceptionForDate(date)
>   adds an exception to the work schedule for given *date*

removeExceptionForDate(date)
>   removes an exception to the work schedule for given *date*

calendarId
>   an internal integer value assigned to each calendar, unique in each document. The standard calendar is alway has a value of 1 (read only)

Workday Type

```
MONDAY_MASK
TUESDAY_MASK
WEDNESDAY_MASK
THURSDAY_MASK
FRIDAY_MASK
SATURDAY_MASK
SUNDAY_MASK
DEFAULT_WORKDAYS
```

Bit values for days of the week. **DEFAULT_WORKDAYS** is the same as the masks for Monday through Friday, OR'd together.

Resource

The Resource class represents the information about a human, capital or production resource that can be assigned to a task. The constructors are not accessible from scripts.

resourceId
> an internal integer value assigned to each resource, unique in each document (read only)

name
> the full name of the resource

abbreviation
> an abbreviation for the resource used to reduce display size

utilization
> a floating point value indication the amount of time the resource is used on the resource. For example a value of 0.5 for a resource with an 8 hour work day only works 4 hours per day on the project. Defaults to 1.0

calendar
> the calendar being used by this resource. Defaults to the standard calendar

GraphElemFilter

GraphElemFilter is a base class for any class used to filter graph elements (instances of GraphElem). The Application class provides six pre-defined instances to filter for entities, junctors, groups, edges, start entities and end entities. Script writers can also create classes derived from GraphElemFilter to perform arbitrary filtering.

filter(elem)
> returns True if the *elem* matches the filter. Derived classes should override this method. Reminder: Your derived class needs to include self as the first parameter, with elem as the second

FontSpec

The FontSpec class represents a specification for a font. The constructor is accessible from scripts.

family
>   the font family; e.g., Arial, New Times Roman, SansSerif, etc. (read only)

style
>   the derived style of the font, plain, bold, italic or bold/italic. Note that fonts like Arial Bold that have an inherent style have this field set to plain (read only)

size
>   the point size of the font. Can be set to AUTOSIZE when the entityTitleFont field of the Document is involved (read only)

>   **PLAIN**
>   constant value indicating a style of plain (read only)

>   **BOLD**
>   constant value indicating a style of bold. This can be or'ed with ITALIC (read only)

>   **ITALIC**
>   constant value indicating a style of italic. This can be or'ed with BOLD (read only)

>   **AUTOSIZE**
>   constant value indicating the size should be determined algorithmically. Only used for the entityTitleFont field of Document (read only)

ascent `API 1.7`
>   the ascent of the font. If you construct a FontSpec at runtime, you must call the calcFont method in a Document instance to make this field valid (read only)

descent `API 1.7`
>   the descent of the font. If you construct a FontSpec at runtime, you must call the calcFont method in a Document instance to make this field valid (read only)

uppercaseHeight `API 1.7`

the maximum ascent of the letters 'X', 'O,' and 'M' of the font. If you construct a FontSpec at runtime, you must call the calcFont method in a Document instance to make this field valid (read only)

init(family, style, size)

creates a new FontSpec with the given *family*, *style* and *size*

GraphOperator

Any object passed to the operate method of Document must be an instance of a class derived from the GraphOperator class.

operate(elem)
> called for each graph element the flags parameter of the Document's operate method. Derived classes should override this method. Reminder: Your derived class needs to include self as the first parameter, with *elem* as the second.

TextEditor

Annotations can be edited via a TextEditor instance instead of having to modify the annotation as a string value, which can be difficult as an annotation is in HTML format. A TextEditor for an annotation is provided via the GraphElem method annotationEditor.

length
　length of the "plain" text, as an integer (read only)

text
　the styled text as an HTML document in a string (read only)

plainText
　the plain text; i.e., the same as text above but with all HTML tags removed (read only)

selection
　the currently selected text as plain text (read only)

selectionStart
　the start of the selection, as an integer

selectionEnd
　the end of the selection, as an integer, always greater than selectionStart

selectionAttributes
　the attributes of the text at selectionStart, as a dictionary of Text Editor Type, see Application class for possible keys and values (read only)

flush( )
　saves all changes to the text, possibly creating an undo record

replace(string, attributes)
　replaces the text between selectionStart and selectionEnd with *string*, assigning the given *attributes* dictionary to the new text, and sets selectionEnd to selectionStart plus the length of string (see the Text Editor Type in Application class for possible attributes keys and values)

insert(string, attributes)
　inserts *string* at selectionStart, assigning the given attributes dictionary to the new text, and sets selectionEnd to selection-

Start plus the length of string (see the Text Editor Type in Application class for possible *attributes* keys and values)

remove( )
> deletes the text between selectionStart and selectionEnd and sets selectionEnd to selectionStart

runStart( )
> searching backwards, finds the first character not matching the attributes at selectionStart starting at selectionStart and returns the index of the next character (as an integer) or zero if the start of the text is reached

runStart(attributes)
> searching backwards, finds the first character not matching the given *attributes* starting at selectionStart and returns the index of the next character (as an integer) or zero if the start of the text is reached

runLimit( )
> searching forward, finds the first character not matching the attributes at selectionStart starting at selectionStart, and returns the index of that character minus the result of calling runStart( ), as an integer

runStart(attributes)
> searching backwards, finds the first character not matching the given *attributes* starting at selectionStart, and returns the index of that character minus the result of calling runStart(attributes), as an integer

runOfSame() `API 1.7`
> returns the number of characters with the same attributes beginning at start of string

runOfSame(start) `API 1.7`
> returns the number of characters with the same attributes beginning at start

reset(string)
> replaces all the text in the editor with *string*, which must be an HTML document

VertexOperator

A VertexOperator is the base class for system operators like Fuzzy And, Fuzzy Or, etc. Instances of VertexOperator are immutable.

name
> user interface long name of the operator; e.g., "Fuzzy And" (read only)

abbreviation
> user interface short name of the operator; e.g., "AND" (read only)

asciiAbbreviation `API 1.7`
> user interface ASCII-compliant short name of the operator; e.g., "AND" (read only)

color `API 1.7`
> the fill color (read only)

CalendarEvent

A Calendar event stores a work schedule exception. Instances of CalendarEvent are immutable.

startDate
> the date of the exception, as a Date (read only)

LayoutLine

Contains position information about one line of text in a title or annotation. This can be used to extract the actual text from a string or rendering attributes from a TextEditor instance.

start
> the index of the first character in this line (read only)

count
> the number of characters in this line (read only)

position
> the baseline position of the first character in the line as a tuple (x, y) (read only)

bounds
> the frame of the characters in this line as a tuple *(x, y, width,*

height) (read only)

Graphic Symbol Names

The getSymbolByName method in the Document class can be used to create an instance of the Symbol class that can be assigned to the symbol field of an entity, group or custom entity class. The format of these names is generator:id; i.e., a generator name and a symbol ID separated by a colon.

For custom symbols generator is either "com.arciem.symbol.SVGSymbolGenerator" for SVG drawings and "com.arciem.symbol.BitmapSymbolGenerator" for bitmap images; e.g., PNG, JPEG, GIF, etc. The id will be a UUID as a string.

The Application object has predefined constants for all the symbols built-in to Flying Logic Pro. One of these constants can be supplied as a parameter to the getSymbolByName method of a Document instance.

Symbol Name Constant	Image
`FLOWCHART_RECTANGLE`	
`FLOWCHART_RECTANGLE_LINED`	
`FLOWCHART_RECTANGLE_ROUND`	
`FLOWCHART_CIRCLE`	
`FLOWCHART_CIRCLE_DOUBLE`	
`FLOWCHART_LOZENGE`	
`FLOWCHART_HEXAGON`	
`FLOWCHART_OVAL`	
`FLOWCHART_PARALLELOGRAM`	

`FLOWCHART_TRAPEZOID_DO`	
`FLOWCHART_SQUARE`	
`FLOWCHART_SQUARE_LINED`	
`FLOWCHART_CRYSTAL`	
`FLOWCHART_PENTAGON`	
`FLOWCHART_OCTAGON`	
`FLOWCHART_TRIANGLE_UP`	
`FLOWCHART_TRIANGLE_DOWN`	
`FLOWCHART_DISPLAY`	
`FLOWCHART_DOCUMENT`	
`FLOWCHART_RECTANGLE_BOW_LEFT`	
`FLOWCHART_RECTANGLE_BOW_OUT`	
`FLOWCHART_SLANT`	
`FLOWCHART_SQUARE_CIRCLE`	
`FLOWCHART_CYLINDER`	
`FLOWCHART_CARD`	

`FLOWCHART_TAPE`	
`FLOWCHART_STRIP`	
`FLOWCHART_CONNECTOR_OFF_PAGE`	
`FLOWCHART_CONNECTOR_ON_PAGE`	
`WRITING_BALLOON_TALK`	
`WRITING_BALLOON_THINK`	
`WRITING_BALLOON_SHOUT`	
`WRITING_BALLOON_DIALOG`	
`DINGBAT_QUESTION_MARK`	
`DINGBAT_CHECK_MARK`	
`DINGBAT_BALLOT_X`	
`DINGBAT_ASTERISK`	
`DINGBAT_ARROW_RIGHT`	
`DINGBAT_STAR`	
`DINGBAT_HAND_RIGHT`	
`DINGBAT_HEART`	

`DINGBAT_LIGHTNING`	
`DINGBAT_INTL_NO`	

Importer and Exporter Examples

Example CSV Importer

CSV Format Description

The Python-based CSV (comma-separated values) importer described here is exactly the same as the one built into Flying Logic.

When using the **File ▶ Import ▶ Import Diagram from CSV...** command, you will first be asked for a text file to import, then you will be presented with two dialogs: The first dialog provides the importer with information on how to read the file (text encoding, whether comma or tab is used as a column delimiter and whether there is a header row) and whether a new document is to be created (instead of importing into the current document).

The second dialog tells the importer how to interpret the columns in the text file. The columns in the text file can either be used as the name of a new entity or group, the entity class of a new entity (ignored from groups), a list of connections to this entity by column number (ignored for groups), a list of children (ignored for entities), or an annotation. A row is considered to represent a group if children is not an empty string of text. The connections can be considered wither a list of predecessors or successors. The valued entered in Internal Separator is the character used to separate column numbers in a connection or children list (defaults to space). The row indexes for predecessors, successors, or children can either start at 1 for the first row, 0 for the first row, or be assigned an index from a value taken from a column.

The entires in a column can be quoted with double quotes. When quoted, appearance of a double quote character in the column text must be escaped with a backslash character.

Example CSV File

Line #	Line Text
1	`"Title","Class","Depends","Group"`
2	`"D","Action",,`
3	`"E","Action",,`
4	`"C","Intermediate Effect","2 3",`
5	`"B","Action",,`
6	`"A","Goal","4 5",`
7	`"G",,,"2 3 4"`

The example above has a header row, comma as separator, uses quoted strings (unnecessary in this case as there are no embedded commas, etc.), and the default list separator of space. Note that the header row is still considered a row, so the "D" entity is row 2.

The example file generates the following diagram:
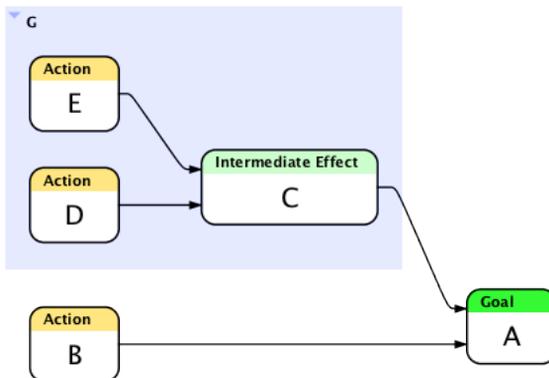
CSV Importer Code

```python
# import_csv.py
# import comma separated values from a file into Flying Logic
# supports comma, tab and semicolon as delimiters and quoted values
# creates entities, groups and edges, but not junctors
# Copyright 2013,2014 Sciral

# Java classes needed to create the UI and read files
from java.io import InputStreamReader, BufferedReader, FileInputStream
from java.nio.charset import Charset
from javax.swing import Box, BoxLayout, JLabel, JCheckBox, JComboBox,
JTextField

# required variable that provides the label for the item in Flying Logic
import menu
importMenuLabel = "Import Diagram from CSV File"

# importLine: a subroutine to process a line
# returns an array of values found in the line
def importLine(line, delimiter):
    cols = []
    stage = 0
    quoted = False
    for c in line:
        if stage == 0:
            if c == '"':
                quoted = True
                stage = 1
                s = []
            elif c == delimiter:
                cols.append('')
            elif not c.isspace():
                quoted = False
                stage = 1
                s = [c]
        elif stage == 1:
            if c == '"':
                if quoted:
                    stage = 2
                else:
                    s.append(c)
            elif c == delimiter:
                if quoted:
                    s.append(c)
                else:
                    cols.append(''.join(s).strip())
                    stage = 0
            else:
                s.append(c)
        elif stage == 2:
            if c == '"':
                s.append(c)
                stage = 1
```

```python
            elif c == delimiter:
                cols.append(''.join(s))
                stage = 0
            elif not c.isspace():
                """ bad format """
                break

if stage != 0:
    cols.append(''.join(s))

    return cols

# this function nicely adds an annotation from plain text
def setAnnotation(elem, text):
    editor = elem.annotationEditor
    editor.insert(text, { })
    editor.flush()

# importDocument: required function for an importer
# parameters
#       file: filename of the file to import
def importDocument(file):
    # create a dialog using Java to collect details about the imported file
    masterBox = Box(BoxLayout.Y_AXIS)

    # text encoding
    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Text Encoding: "))
    encodingsComboBox = JComboBox(['Windows/Latin-1/ISO-8859-1', 'UTF-8',
'ASCII (US)'])
    controlBox.add(encodingsComboBox)
    masterBox.add(controlBox)

    # delimiter
    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Column separator: "))
    columnSepComboBox = JComboBox(["Commas", "Tabs", "Semicolon"])
    controlBox.add(columnSepComboBox)
    masterBox.add(controlBox)

    # header row?
    controlBox = Box(BoxLayout.X_AXIS)
    headerCheckbox = JCheckBox("Has header row")
    controlBox.add(headerCheckbox)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    # create new document? (default is to import into current document)
    controlBox = Box(BoxLayout.X_AXIS)
    newDocCheckbox = JCheckBox("Create new document")
    controlBox.add(newDocCheckbox)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)
```

```python
    # display dialog and collect options
    if 0 == Application.request("CSV Import Settings", masterBox, ("Cancel",
"OK")):
        return

    createNewDocument = newDocCheckbox.isSelected()

knownEncodings = ['ISO-8859-1', 'UTF-8', 'US-ACSII']
    encoding = knownEncodings[encodingsComboBox.selectedIndex]

    columnDelimiter = ','
    if columnSepComboBox.selectedIndex == 1:
        columnDelimiter = '\t'
if columnSepComboBox.selectedIndex == 2:
    columnDelimiter = ';'

    hasHeader = headerCheckbox.isSelected()

    theDoc = document
    if createNewDocument:
        theDoc = Application.newDocument()

    # open input file
    reader = BufferedReader( InputStreamReader( FileInputStream( file ),
encoding ) )

firstLine = True
    vertexList = []
    indexMap = {}
    row = 0

    # process file line by line
    while True:
        line = reader.readLine()
        if line == None:
            break

        row = row + 1

        # collect values in line
        columns = importLine(line, columnDelimiter)
        numColumns = len(columns)

        # if first line, ask user to identify meaning if each column
        if firstLine:
            firstLine = False

            columnNames = ['Not used']
            indexNames = ['First row is index 1', 'First row is index 0']
            if hasHeader:
                columnNames = columnNames + columns
                indexNames = indexNames + columns
            else:
                for i in range(numColumns):
```

```python
            columnNames.append('Column ' + str(i + 1))
            indexNames.append('Column ' + str(i + 1))

    # make Java dialog
    masterBox = Box(BoxLayout.Y_AXIS)

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Please match attributes with columns:"))
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    masterBox.add(Box.createVerticalStrut(20))

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Element Title: "))
    titleColumnComboBox = JComboBox(columnNames)
    controlBox.add(titleColumnComboBox)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Entity Class: "))
    classColumnComboBox = JComboBox(columnNames)
    controlBox.add(classColumnComboBox)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Connections: "))
    linkColumnComboBox = JComboBox(columnNames)
    controlBox.add(linkColumnComboBox)
    predColumnComboBox = JComboBox(['Predecessors', 'Successors'])
    controlBox.add(predColumnComboBox)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Children: "))
    childColumnComboBox = JComboBox(columnNames)
    controlBox.add(childColumnComboBox)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Internal separator: "))
    rowSepTextField = JTextField(5)
    controlBox.add(rowSepTextField)
    controlBox.add(Box.createHorizontalGlue())
    masterBox.add(controlBox)

    controlBox = Box(BoxLayout.X_AXIS)
    controlBox.add(JLabel("Row Index: ")),
    indexColumnComboBox = JComboBox(indexNames)
    controlBox.add(indexColumnComboBox)
```

```
            controlBox.add(Box.createHorizontalGlue())
            masterBox.add(controlBox)

            controlBox = Box(BoxLayout.X_AXIS)
            controlBox.add(JLabel("Annotation: "))
            noteColumnComboBox = JComboBox(columnNames)
            controlBox.add(noteColumnComboBox)
            controlBox.add(Box.createHorizontalGlue())
            masterBox.add(controlBox)

            if 0 == Application.request("CSV Column Interpretation",
masterBox, ("Cancel", "OK")):
                if createNewDocument:
                    theDoc.closeDocument(False)
                return

            titleColumn = titleColumnComboBox.selectedIndex – 1

            classColumn = classColumnComboBox.selectedIndex – 1

            linkColumn = linkColumnComboBox.selectedIndex – 1

            childrenColumn = childColumnComboBox.selectedIndex – 1

            isSuccessor = (predColumnComboBox.selectedIndex == 1)

            indexColumn = indexColumnComboBox.selectedIndex – 2

            noteColumn = noteColumnComboBox.selectedIndex – 1

            rowDelimiter = rowSepTextField.text.strip()
            if len(rowDelimiter) == 0:
                rowDelimiter = ' '

            # if first line is a header, skip line
            if hasHeader:
                continue

    # default entity and group attributes
    entityTitle = 'untitled'
        entityClass = 'Generic'
        entityLinks = ''
        groupChildren = ''
        annotation = None

        # match values with identified attributes
        if titleColumn >= 0 and titleColumn < numColumns:
            entityTitle = columns[titleColumn]
if classColumn >= 0 and classColumn < numColumns:
    entityClass = columns[classColumn]
        if linkColumn >= 0 and linkColumn < numColumns:
            entityLinks = columns[linkColumn]
    if childrenColumn >= 0 and childrenColumn < numColumns:
        groupChildren = columns[childrenColumn]
```

```python
            if noteColumn >= 0 and noteColumn < numColumns:
                annotation = columns[noteColumn]
                if len(annotation) == 0:
                    annotation = None

    # create index mapping based on user choice of one-based, zero-based or by
    column value
    indexRow = row
        if indexColumn >= 0:
            indexRow = int(columns[indexColumn])
            elif indexColumn == -1:
                indexRow = row - 1
        indexMap[indexRow] = row

            # either handle as group or entity -- no junctors yet
            if groupChildren != '':
                group = theDoc.newGroup(None)[0]
                if entityTitle != 'untitled':
                    group.title = entityTitle
                if annotation != None:
                    setAnnotation(group, annotation)

                vertexList.append( (group, None, groupChildren) )
            else:
                entity = theDoc.addEntityToTarget(None)[0] # no need to
    clearSelection each iteration
                entity.title = entityTitle

                eCls = theDoc.getEntityClassByName(entityClass)
                if eCls != None:
                    entity.entityClass = eCls
                if annotation != None:
                    setAnnotation(entity, annotation)

                vertexList.append( (entity, entityLinks, None) )

    # generate new elements from collected vertex data
    for data in vertexList:
        if data[1] != None:
            predList = data[1].split(rowDelimiter)
                for pred in predList:
                    if len(pred) > 0:
                        index = indexMap[int(pred)] - 1
                        if hasHeader:
                            index = index - 1
                        if index >= 0 and index < len(vertexList):
                            if isSuccessor:
                                theDoc.connect(data[0], vertexList[index][0])
                            else:
                                theDoc.connect(vertexList[index][0], data[0])
        if data[2] != None:
            childList = data[2].split(rowDelimiter)
                for child in childList:
                    if len(child) > 0:
```

```
                    index = indexMap[int(child)] – 1
                    if hasHeader:
                        index = index – 1
                    if index >= 0 and index < len(vertexList):
                        vertexList[index][0].parent = data[0]

        return theDoc
```

Example DOT Exporter

The code below exports a document to a DOT (GraphViz) file, but with
less features then the native export option in Flying Logic.

```python
# export_dot.py
# a simple DOT format exporter, less complete then the native version in
Flying Logic
# Copyright 2013 Sciral

# required variable that provided the label for the item in Flying Logic
export menu
exportMenuLabel = "Export Diagram to simple DOT format"

# exportDocument: required function for an exporter
# parameters
# file: filename of the file to export
def exportDocument(file):
    # open output file using Python file I/O
    fh = open(file, 'w')

    fh.write("digraph graphname {\n")
    for elem in document.all:
        if elem.isGroup or elem.isEdge:
            continue

        # use the element unique id's (eid) to create unique id's in DOT
        if elem.isEntity:
            fh.write("\tn" + str(elem.eid) + " [label=\"" + elem.title +
"\"];\n")
        if elem.isJunctor:
            fh.write("\tn" + str(elem.eid) + " [label=\"" + elem.operator.
abbreviation + "];\n")
        for outEdge in elem.outEdges:
            fh.write("\tn" + str(elem.eid) + " –> n" + str(outEdge.target.
eid) + ";\n")
            fh.write("\t}\n")

fh.close()
```

Flying Logic Document Format

Flying Logic documents are XML-formatted files. The schema for release 3 documents can be downloaded at http://flyinglogic.com/XMLSchema/flyinglogic-3.xsd. Following are some supporting tables explaining more about the schema.

Reference Tables

The table below gives information about each XML element that can be found in a Flying Logic document. The elements are listed in the order they generally appear in a document. The `attributes` and `attribute` elements are common children of many elements, so those elements and their children are documented at the end of this table.

Because the words "attributes" and "attribute" are used as element names in a Flying Logic document, those words will be shown in a `fixed-width font` when referring to the element names and not as an XML attribute value.

Element	Description
`flyingLogic`	The root element. Attributes are `majorversion` (required, "3" for Flying Logic 2.X), `minorversion` and `uuid`. Can contain `domains`, `symbols`, `displaySettings`, `canvasSettings`, `inspectorSettings`, `printSettings`, `documentInfo` and `decisionGraph` elements. During XSLT export will also contain an `exportDocumentInfo` element.
`domains`	The custom domains defined in the document. Can contain `domain` and `alteredBuiltinDomain` elements.
`domain`	A custom domain. Contains `entityclass` and `attributes` elements.
`entityclass`	Either a custom entity class in a domain or a reference to an entity class as an entity `attribute`. For a custom entity class, contain an `attributes` element. For a reference, has required XML attributes name and `uuid`.

Element	Description
`alteredBulltinDomain`	Appears when a built-in domain is hidden. Required attributes are `domainName` and `entityClass` (the guid of one entity class in the domain). Contains an `attributes` element.
`symbols`	The custom symbols defined in this document. Can contain `symbol` elements.
`symbol`	Either a custom symbol or a reference to a symbol as an entity or group `attribute`. Required attributes are `generator` and `idCode`. A custom symbol definition will contain either additional elements or text depending on the format of the custom symbol, and can have a `clip` element. Bitmap image symbols contain a base-64 encoded dump of a PNG file as text.
`clip`	A clip rectangle for a symbol. If missing, the default is the whole "image." Required attributes are `x`, `y`, `width` and `height`.
`data`	Appears in an SVG symbol. Contains a CDATA with the XML text from an SVG file.
`displaySettings`	The visual elements and layout of the graph. All the attributes are optional: `addEntityAsSuccessor`, `annotationToBrowse`, `bias`, `confidenceVisible`, `edgeNotesVisible`, `edgeWeightsVisible`, `entityIdVisible`, `noteNumbersVisible`, `orientation`, and `projectManagementVisible`.
`canvasSettings`	The scroll position and zoom level of the graph. Optional attributes are `horizScroll`, `vertScroll` and `zoom`.

Element	Description
`chartSettings`	The column visibility and with settings for chart view. All the attributes are optional: `classVisible`, `classWidth`, `completionVisible`, `effortVisible`, `finishDateVisible`, `resourceVisible`, `resourceWidth`, `startDateVisible`, and `titleWidth`.
`inspectorSettings`	This element has the state of the sidebar (inspector panel) and contains `inspector` elements. The attributes are `visibility` and `width`.
`inspector`	State information about an inspector. The `title` attribute matches the internal name of one inspector. The optional attributes are `visibility` and `height`. Only the Domain, Text and User-Defined Attributes inspectors have a height.
`printSettings`	Saved printing settings for the document. Optional attributes are `footerLeft`, `footerMiddle`, `footerRight`, `headerLeft`, `headerMiddle`, `headerRight`, `paperMargins`, `paperName`, `paperOrientation`, `paperSize`, `saveInk` and `showselhalo`. (Yes, `showselhalo` is all lowercase.)
`documentInfo`	Has attributes for some of the values settable the Document inspector: `author`, `comments`, `keywords` and `title`.
`decisionGraph`	The actual graph. Contains `entityOperator`, `defaultJunctorOperator` and `logicGraph` elements.

Element	Description
`entityOperator`	The current entity operator for the graph. The `class` attribute is the name of a vertex operator. See notes about vertex operators at after this table.
`defaultJunctorOperator`	The current default junctor operator for newly created junctors. The `class` attribute is the name of a vertex operator. See notes about vertex operators at after this table.
`calendars`	The calendars defined in the document. Can be missing if project management has never been enabled for the document. Contains one or more indexed element elements that themselves contain a workdays element. See `com.flyinglogic.app.project.Workweek` in table below for details.
`resources`	The resources defined in the document. Can be missing if no resources are defined. Contains one or more indexed element elements that themselves contain a resource element. See `com.flyinglogic.app.project.Resource` in table below for details.
`logicGraph`	The organization of the graph. Contains a `operatorFamily` and `graph` element.
`operatorFamily`	The key-value family of the graph. Has one required attribute `class` with a required value of `com.flyinglogic.decisiongraph.DecisionGraphOperatorFamily.`

Element	Description
graph	The elements of the graph. Contains **attributes**, **vertices** and **edges** elements.
vertices	Contains **vertex** elements representing entities, junctors and groups.
vertex	An entity, junctor or group. Has a required attribute of **eid**, an unique integer assigned to each graph element. Entities and junctors have optional attributes of **inEdgeOrder** and **outEdgeOrder**, which are space-delimited list of **eid** values for edges connected to the element. Groups have a **grouped** attribute, a space-delimited list of **eid** values for child vertices, and an optional **collapsed** attribute which has a value of **true** is the group is collapsed. Contains an **attributes** element. During XSLT export can also contain an **exportAttributes** element.
edges	Contains **edge** elements.
edge	An edge. Has three required attributes: **eid**, an unique integer assigned to each graph element; and **source** and **target**, the **eid** of the connected vertices. During XSLT export can also contain an **exportAttributes** element.
attributes	Contains **attribute** elements.
attribute	An **attribute** is a key-value pair with an associated data type for the value. There are two required attributes, **key** and **class**. Each known **class** has its own format for its contents. See the table of recognized classes below.

Element	Description
`extendedDocumentInfo`	This element only appears during XSLT export. Has attributes `title` (the document file name as a title), `screenWidth` and `screenHeight` (visible dimensions of graph) and `hoistKey` (eid of hoisted group if graph hoisted).
`exportAttributes`	This element only appears during XSLT export if the flags XSLT_INCLUDE_FRAMES or XSLT_INCLUDE_EDGE_SPLINES are set in the exportDocument call. Contains `exportAttribute` elements.
`exportAttribute`	A key-value pair with a `key` attribute and value of either text content or child elements. If XSLT_INCLUDE_FRAMES flag is set and a graph element is visible. it will have exportAttribute elements with values for `x`, `y`, `width` and `height`. If XSLT_INCLUDE_EDGE_SPLINES is set, a visible edge will have an exportAttribute with a `key` of "path" and contains a `bezierList` element.
`bezierList`	Has attributes of `head` and `tail` that indicate what the edge is connected to: `source`, `target`, `annotation`, `weight`, `entityhoistcap` or `junctorhoist-cap`. Contains one or more `bezier` elements.
`bezier`	Has attributes giving the control points `x1`, `y1`, `x2`, `y2`, `x3`, `y3`, `x4` and `y4`.

The contents of an **attribute** element varies by the value of the **class** attribute. The first table below lists all the know keys, the class for that key and the allowed parent element of the **attributes** element it appears within. The second table describes the contents for each recognized class. Operators have more details after these two tables.

Value of `key`	Value of `class`	Parent elements (notes)

`user.identifier`	`java.lang.String`
	`java.lang.Integer`
	`java.lang.Double`
	`java.lang.Boolean`
		`graph`
		`vertex`
		`edge`
		(user defined attribute where *identifier* is the string appearing in the UI or key to use with the **user** array in scripts)

`title`	`java.lang.String`	**vertex** (entity and group only)
`confidence`	`com.arciem.` `FuzzyBoolean`	**edge** **vertex** (entity only)
`type`	`java.lang.String`	**vertex** (either "entity" or "junctor")
`typeid`	`java.lang.Integer`	**vertex** (entity only)
`completion`	`java.lang.Double`	**vertex** (entity only)

`effortTime`	`com.flyinglogic.`
	`app.project.`
	`FLTimeInterval`
		vertex (entity only)

Value of `key`	Value of `class`	Parent elements (notes)
~~effort~~	`java.lang.Double`	`entityClass` `vertex` (entity only) (1.0 equals one day) *deprecated. Do not use in 3.0 or later documents.*
`startDate`	`com.flyinglogic.app.` `project.FLDate`	`vertex` (entity only)
`endDate`	`com.flyinglogic.app.` `project.FLDate`	`vertex` (entity only)
`projectStartDate`	`com.flyinglogic.app.` `project.FLDate`	`graph` (calculated with finish-to-start scheduling)
`projectEndDate`	`com.flyinglogic.app.` `project.FLDate`	`graph` (calculated with start-to-finish scheduling)
`resAssignment`	`java.lang.String`	`vertex` (entity only)
`startTime`	`com.flyinglogic.` `app.project.` `FLTimeInterval`	`vertex` (entity only, always calculated for now)
`endTime`	`com.flyinglogic.` `app.project.` `FLTimeInterval`	`vertex` (entity only, always calculated for now)
`utilization`	`java.lang.Double`	`vertex` (entity only, informational only)
`noteHTML`	`com.arciem.` `CDATAElement`	`edge` `vertex` (styled annotation as XHTML text)

Value of `key`	Value of `class`	Parent elements (notes)
`note`	`java.lang.String`	`edge` `vertex` (unformatted version of noteHTML)
`noteNumber`	`java.lang.Integer`	`edge` `vertex`
`entityClass`	`com.flyinglogic.` `logicgraph.` `entityclass.` `EntityClass`	`vertex` (entity only)
`symbol`	`com.arciem.symbol.` `Symbol`	`entityClass` `vertex` (entity and group only)
`weight`	`java.lang.Double`	`edge`
`backEdge`	`java.lang.Boolean`	`edge`
`forwardOperator`	*operator* (for `edge` always `com.flyinglogic.` `logicgraph.operator.` `MultiplyEdgeOperator`)	`edge` `vertex` (entity and junctor only)
`color`	`com.arciem.graphics.` `ColorRGB`	`entityClass` `vertex` (group only)
`name`	`java.lang.String`	`domain` `entityClass`
`uuid`	`java.util.UUID`	`entityClass`
`builtin`	`java.lang.Boolean`	`domain` `entityClass`
`hidden`	`java.lang.Boolean`	`domain` `entityClass`
`edgeWeight`	`java.lang.Double`	`entityClass`
`showname`	`java.lang.Boolean`	`entityClass`

Value of `class`	Description of `attribute` contents
`java.lang.String`	Text is a string.
`java.lang.Integer`	Text is an integer.
`java.lang.Double`	Text is a real.
`java.lang.Boolean`	Text is either `true` or `false`.
`java.util.UUID`	Text is a text-encoded globally unique identifier (GUID).
`com.arciem.CDATAElement`	A CDATA node containing XML or XHTML.
`com.arciem.FuzzyBoolean`	A `fuzzyBoolean` element with a required attribute of `true` with a real value normally between 0.0 and 1.0.
`com.flyinglogic.app.project.FLDate`	A `date` element with a required attribute `value` of a ISO-8601 date and time in UTC. The time should always be 00:00:00 in this version of Flying Logic.
`com.arciem.symbol.Symbol`	A `symbol` element (see above). Can have just an `idCode` of "none", indicating the default symbol for an entity class is being hidden.
`com.flyinglogic.logicgraph.entityclass.EntityClass`	An `entityClass` element (see above).
`com.arciem.graphics.ColorRGB`	A colorRGB element with three attributes `red`, `green` and `blue` as real values between 0.0 and 1.0.

Value of `class`	Description of `attribute` contents
`com.flyinglogic.app.project.Workweek`	A `workdays` element. Has three required attributes of `rid` (resource ID), `days` (a comma-delimited days of the week as lower-case three-letter abbreviations) and `hours` (must be 1.0 to 23.0). Can contain `event` elements that act as exceptions to the work schedule.
`com.flyinglogic.app.project.Resource`	a resource element. Has three required attributes: `rid` (resource ID), `name`, and `cid` (calendar ID). Can contain `abbr` (abbreviation, can be empty) and `utilization` (value between 0.0 and 1.0, but not 0.0) attributes.
`com.flyinglogic.app.project.FLTimeInterval`	an interval element. Contains an ISO-8601 time value; e.g., `PT8H0M0S` represents a time interval of 8 hours

Each kind of vertex and edge operator has a unique contained element that has no attributes or contents. The table below lists the single element contained in each operator by `class`.

Operator class	Contained element
`com.flyinglogic.logicgraph.operator.FuzzyOrVertexOperator`	`fuzzyOr`
`com.flyinglogic.logicgraph.operator.FuzzyAndOperator`	`fuzzyAnd`
`com.flyinglogic.logicgraph.operator.FuzzyXorVertexOperator`	`fuzzyXor`
`com.flyinglogic.logicgraph.operator.AverageVertexOperator`	`average`
`com.flyinglogic.logicgraph.operator.ComplementVertexOperator`	`comp`

Operator class	Contained element
`com.flyinglogic.logicgraph.operator.` `DistributorVertexOperator`	`dist`
`com.flyinglogic.logicgraph.operator.` `MaxVertexOperator`	`max`
`com.flyinglogic.logicgraph.operator.` `MinVertexOperator`	`min`
`com.flyinglogic.logicgraph.operator.` `MultiplyVertexOperator`	`multiply`
`com.flyinglogic.logicgraph.operator.` `NegateVertexOperator`	`neg`
`com.flyinglogic.logicgraph.operator.` `ProductVertexOperator`	`product`
`com.flyinglogic.logicgraph.operator.` `ProportionVertexOperator`	`proportion`
`com.flyinglogic.logicgraph.operator.` `ReciprocalVertexOperator`	`reciprocal`
`com.flyinglogic.logicgraph.operator.` `SumVertexOperator`	`sum`
`com.flyinglogic.logicgraph.operator.` `SumProbabilitiesVertexOperator`	`sump`
`com.flyinglogic.logicgraph.operator.` `MultiplyEdgeOperator`	`multiply`