SIMULIA

3DEXPERIENCE

ABAQUS 2026 FD04
GUI TOOLKIT USER'S GUIDE

DASSAULT SYSTEMES

Contents

Trademarks and Legal Notices....4
Abaqus GUI Toolkit User's Guide....5
What's New....6
Introduction....7
What can I do with the Abaqus GUI Toolkit?....8
Prerequisites for using the Abaqus GUI Toolkit....9
Abaqus GUI Toolkit basics....10
Organization of the Abaqus GUI Toolkit User's Guide....11
Getting Started with the Abaqus GUI Toolkit....13
The kernel and GUI....14
What are the components of an Abaqus GUI application?....15
Plug-ins and customized applications....17
Running the prototype application....18
Building Dialog Boxes....20
Widgets....21
Labels and buttons....22
Text widgets....33
Lists and combo boxes....37
Range widgets....41
Tree widgets....44
Table widget....48
Miscellaneous widgets....67
The create method....72
Widgets and fonts....73
Layout managers....74
About layout managers....75
Padding and spacing....76
Horizontal and vertical frames....77
Vertical alignment for composite children....78
General-purpose layout managers....79
Row and column layout manager....80
Resizable regions....81
Rotating regions....82
Tab books....83
Layout hints....85
Layout examples....86
Tips for specifying layout hints....89
Dialog boxes....90
About dialog boxes....91
Modal versus modeless....92
Showing and hiding dialog boxes....93
Message dialog boxes....94
Custom dialog boxes....100

Data dialog boxes....108
Common dialog boxes....118
Issuing Commands....123
Commands....124
An overview of commands....125
The kernel and GUI processes....126
Executing commands....127
Kernel commands....128
GUI commands....129
AFXTargets....150
Accessing kernel data from the GUI....151
Receiving notification of kernel data changes....152
Modes....157
About modes....158
Mode processing....159
Form modes....168
Procedure modes....174
Picking in procedure modes....181
GUI modules and toolsets....193
Creating a GUI module....194
Examining a GUI module example....195
Registering a GUI module....206
Switching to a GUI module....207
Creating a GUI toolset....208
GUI Toolset example....209
Creating toolset components....210
Registering toolsets....211
Customizing an existing module or toolset....212
Modifying and accessing Abaqus/CAE GUI modules and toolsets....213
The File toolset....218
The Tree toolset....219
The Selection toolset....220
The Help toolset....221
An example of customizing a toolset....222
Creating a customized application....224
Creating an application....225
About the GUI code....226
Startup script....227
Licensing and execution....229
The application object....230
The application object....231
Common methods....232
The main window....233
About the main window....234
The title bar....236
The menu bar....237
Toolbars....238

The context bar....239
The module toolbox....240
The drawing area and canvas....241
The prompt area....242
The message area....243
The command line interface....244
Customizing the main window....245
Modules and toolsets....246
The Abaqus/CAE main window....247
Icons....253
Colors and RGB values....254
Layout hints....272

Trademarks and Legal Notices

Trademarks

Abaqus, **3D**EXPERIENCE®, the 3DS logo, the Compass icon, IFWE, 3DEXCITE, 3DVIA, BIOVIA, CATIA, CENTRIC PLM, DELMIA, ENOVIA, GEOVIA, MEDIDATA, NETVIBES, OUTSCALE, SIMULIA and SOLIDWORKS are commercial trademarks or registered trademarks of Dassault Systèmes, a European company (Societas Europaea) incorporated under French law, and registered with the Versailles trade and companies registry under number 322 306 440, or its subsidiaries in the United States and/or other countries. All other trademarks are owned by their respective owners. Use of any Dassault Systèmes or its subsidiaries trademarks is subject to their express written approval.

Legal Notices

Abaqus and this documentation may be used or reproduced only in accordance with the terms of the software license agreement signed by the customer, or, absent such agreement, the then current software license agreement to which the documentation relates.

This documentation and the software described in this documentation are subject to change without prior notice.

Dassault Systèmes or its Affiliates shall not be responsible for the consequences of any errors or omissions that may appear in this documentation.

© Dassault Systèmes Americas Corp., 2026.

For a full list of the third-party software contained in this release, please go to the Legal Notices in the Abaqus 2026 HTML documentation, which can be obtained from a documentation installation, or in the SIMULIA Established Products 2026 Program Directory, which is available on *www.3ds.com*.

Abaqus GUI Toolkit User's Guide

The Abaqus GUI Toolkit User's Guide describes the Abaqus GUI Toolkit, which allows you to customize the Abaqus/CAE Graphical User Interface by creating your own dialog boxes, GUI modules, and even applications. The guide is designed to guide you through the process of writing an application by explaining how to use the components of the toolkit and by providing snippets of example code.

This guide is a part of the Abaqus® documentation collection, which describes all the capabilities of the Abaqus finite element analysis technology used in SIMULIA® applications.

What's New

This page describes recent changes in Abaqus GUI Toolkit.

2025 FD01

PDF Guides Available

You can download PDF versions of the Abaqus guides from each guide overview.

Benefits: You can easily access the PDF versions of the guides.
In earlier releases, PDF versions of the guides were updated only for the GA (General Availability) release and could be downloaded from the Dassault Systèmes Knowledge Base. As of 2025 FD01, the PDF guides will be updated with each release of the HTML versions of the guides.

For more information, see *Abaqus GUI Toolkit User's Guide*.

Introduction

This chapter provides an overview of the Abaqus GUI Toolkit.
The Abaqus GUI Toolkit is one of the Abaqus Process Automation tools that allow you to modify and extend the capabilities of the Abaqus/CAE graphical user interface (GUI) to enable a wide range of users to generate more efficient Abaqus solutions.

In this section:

- *What can I do with the Abaqus GUI Toolkit?*
- *Prerequisites for using the Abaqus GUI Toolkit*
- *Abaqus GUI Toolkit basics*
- *Organization of the Abaqus GUI Toolkit User's Guide*

What can I do with the Abaqus GUI Toolkit?

You can use the Abaqus GUI Toolkit to customize Abaqus products.

There are many ways to customize Abaqus products.

- User subroutines allow you to change the way Abaqus/Standard and Abaqus/Explicit compute analysis results. Information on user subroutines can be found in the *Abaqus User Subroutines Guide*.
- Environment files allow you to change various default settings. Information on environment files can be found in the *Abaqus Execution Guide*.
- Kernel scripts allow you to create new functions to perform modeling or postprocessing tasks. Information on kernel scripts can be found in the *Abaqus Scripting User's Guide*.
- GUI scripts allow you to create new graphical user interfaces. GUI scripts are described in this guide.

The Abaqus GUI Toolkit provides programming routines that allow you to create or modify components of the GUI. The toolkit allows you to do the following:

- Create a new GUI module. A GUI module is a grouping of similar functionality, such as the Part module in Abaqus/CAE.
- Create a new GUI toolset. A GUI toolset is similar to a GUI module in that it groups similar functionality, but it generally contains more specific functionality that may be used by one or more GUI modules. The Datum tools in Abaqus/CAE are an example of a GUI toolset.
- Create new dialog boxes. The Abaqus GUI Toolkit provides a full library of widgets from which you can construct your own dialog boxes. However, the Abaqus GUI Toolkit is not designed to allow you to modify existing Abaqus/CAE dialog boxes.
- Remove Abaqus/CAE GUI modules and toolsets. You can choose which GUI modules to include in your application and which GUI modules to omit. For example, the Abaqus/Viewer application does not include the modeling-related GUI modules; it contains only the Visualization module.
- Remove some top-level menus or some items in those top-level menus. For example, you could remove the entire top-level **Viewport** menu to prevent users from manipulating viewports, or you could remove the **Import** and **Export** menu items from the **File** menu.
- Perform limited changes to Abaqus/CAE GUI modules and toolsets. For more information, see *Modifying and accessing Abaqus/CAE GUI modules and toolsets*.

The Abaqus GUI Toolkit is not designed to run outside of Abaqus/CAE; it must be used with Abaqus/CAE in order for the infrastructure to function properly.

Prerequisites for using the Abaqus GUI Toolkit

To write applications using the Abaqus GUI Toolkit, you need to have some experience in Python programming, Abaqus kernel commands, object-oriented programming, and GUI design.

Python programming

You should have some experience with the Python language before you write Abaqus/CAE kernel scripts. You should have similar experience when you program GUI applications.

Abaqus kernel commands

The ultimate goal of the GUI is to send a command to the kernel for execution; therefore, you should understand how kernel commands work.

Object-oriented programming

Python is an object-oriented language, and writing an application generally consists of deriving your own new classes, writing methods for them, and manipulating their data. You should understand the concepts of object-oriented programming.

GUI design

Depending on the complexity of your application, it may be helpful to have some training in user-interface design and usability testing. This will help you create an application that is both intuitive and easy to use.

Abaqus offers training classes that cover Python, kernel scripting, and GUI design. For more information, contact your local sales office. Training in GUI design is also available from a number of independent training organizations.

Abaqus GUI Toolkit basics

The Abaqus GUI Toolkit is an extension of the FOX GUI Toolkit, just as the Abaqus Scripting Interface is an extension of the Python programming language.
FOX, which stands for Free Objects for X, is a modern, object-oriented, platform-independent GUI toolkit. Since the Abaqus GUI Toolkit is platform-independent, once you write an application for one platform, you can run that application on all supported platforms—you do not need to change your source code.

The user interface produced by the Abaqus GUI Toolkit looks similar on all platforms. This is due to the architecture of the toolkit. While the application programming interface (API) is the same on all platforms, the underlying calls made to the operating system's GUI libraries differ—on Linux systems calls are made to the `Xt` library, whereas on Windows systems calls are made to the `Win32` library. These differences are hidden from the application developer.

Since the FOX GUI Toolkit is object oriented, it allows developers to extend its functionality easily by deriving new classes from the base toolkit. The Abaqus GUI Toolkit takes advantage of this feature by adding special functionality required for Abaqus GUIs. Class names that begin with `FX` are part of the standard FOX library; for example, `FXButton`. Class names that begin with `AFX` are part of the Abaqus extensions to the FOX library; for example, `AFXDialog`. When the same class exists with both the `FX` and `AFX` prefix (for example, `FXTable` and `AFXTable`), you should use the `AFX` version since it provides enhanced functionality for building applications using Abaqus.

Organization of the Abaqus GUI Toolkit User's Guide

This guide is organized by functionality and is designed to guide developers through the process of writing an application by explaining how to use the components of the toolkit and by providing snippets of example code.
A separate *Abaqus GUI Toolkit Reference Guide* that contains an alphabetical listing of all of the toolkit calls is provided.

The Abaqus GUI Toolkit is based on the FOX GUI toolkit. While this guide explains some of the basic concepts of the FOX toolkit, it does not provide details for many other aspects of the FOX toolkit. For more details on the FOX GUI toolkit, refer to the FOX web site (*http://www.fox-toolkit.org*).

This guide consists of the following sections:

Widgets

This section describes some of the most commonly used widgets in the Abaqus GUI Toolkit.

Layout managers

This section describes how to use the various layout managers in the Abaqus GUI Toolkit to arrange widgets in a dialog box.

Dialog boxes

This section describes the dialog boxes that you can create using the Abaqus GUI Toolkit.

Commands

In an application that employs a graphical user interface, the interface must collect input from the user and communicate that input to the application. In addition, the graphical user interface must keep its state up-to-date based on the state of the application. This section describes how those tasks are accomplished using the Abaqus GUI Toolkit and the two types of commands in Abaqus/CAE—kernel commands and GUI commands.

Modes

A mode is a mechanism for gathering input from the user, processing that input, and then issuing a command to the kernel. This section describes the modes that are available in the Abaqus GUI Toolkit.

Creating a GUI module

This section describes how you can create a GUI module.

Creating a GUI toolset

This section describes how you can create a GUI toolset.

Customizing an existing module or toolset

The previous sections describe how you can create a new module or toolset. Alternatively, the Abaqus GUI Toolkit allows you to derive a new module or toolset from an existing module or toolset and to add or remove functionality from it.

Creating an application

This section explains how to create an application, such as Abaqus/CAE. It also describes the high-level infrastructure that is responsible for running the application.

The application object

This section describes the Abaqus application object. The application object manages the message queue, timers, chores, GUI updating, and other system facilities.

The main window

This section describes the layout, components, and behavior of the Abaqus main window.

Customizing the main window

The main window base class provides the GUI infrastructure to allow user interaction, the manipulation of modules, and the display of objects in the viewport. This section describes how you add functionality to an application by deriving from the main window base class and then registering modules and toolsets.

Getting Started with the Abaqus GUI Toolkit

This chapter provides an overview of a customized GUI application.

In this section:

- *The kernel and GUI*
- *What are the components of an Abaqus GUI application?*
- *Plug-ins and customized applications*
- *Running the prototype application*

The kernel and GUI

Abaqus/CAE executes in two separate processes: the kernel and the GUI.

The role of the kernel is to provide access to Abaqus databases and the commands that create and modify those databases. The role of the GUI is to collect user input, which is then packaged as a command string and sent to the kernel for execution. The GUI is not essential to the execution of Abaqus—an entire model can be constructed, analyzed, and post-processed through the use of kernel scripts, without ever invoking the GUI.

Typically, when you develop some custom functionality you start by creating the kernel commands that implement that functionality. These commands can be debugged by executing them from the command line interface (CLI) in Abaqus/CAE. Once you have determined that the kernel commands are working correctly from the CLI, you can design a GUI to collect the user inputs needed by your commands.

What are the components of an Abaqus GUI application?

There are many components involved in creating a GUI application.
Figure 1 shows an overview of these components and how they are connected. This section provides a brief overview of each component. The components are discussed in more detail in subsequent chapters.

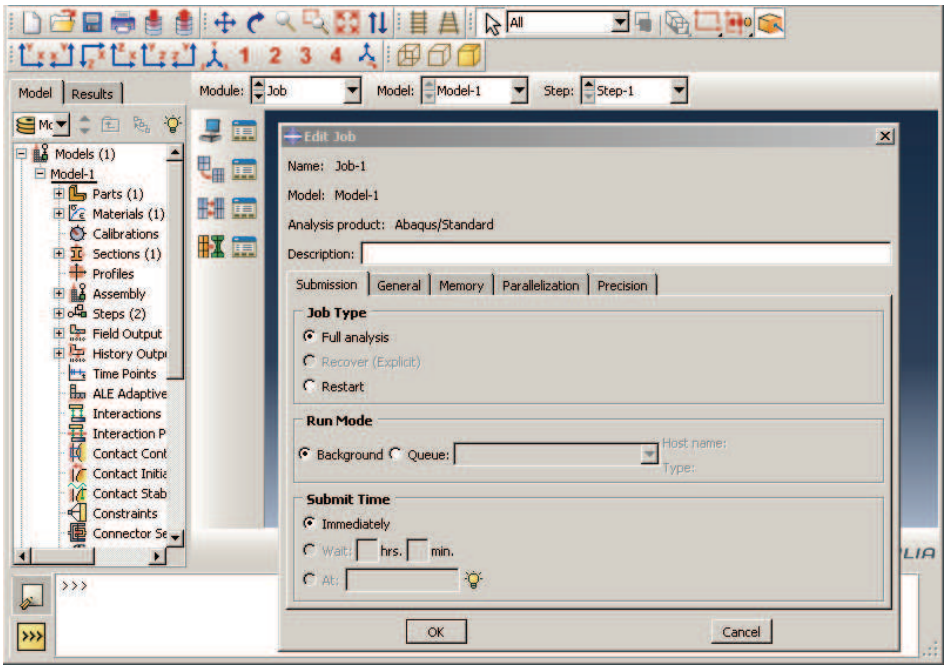


Figure 1: An overview of an Abaqus GUI application.

Widgets

At the lowest level of an application, you use widgets to collect input from the user through a graphical user interface. For example, a text field widget presents a box into which the user can type numbers. Similarly, a check button widget presents a small box that the user can click on to toggle an option on or off.

Layout managers

Layout managers arrange widgets by providing alignment options. For example, a horizontal frame arranges widgets in a row. A vertical frame arranges widgets in a column.

Dialog boxes

Dialog boxes group widgets inside layout managers and present all the inputs required for a particular function. For example, the **Print** dialog box presents all the controls that allow the user to specify what should be printed and how it should be printed.

Modes

Modes are GUI mechanisms that control the display of a particular user interface. Modes are also responsible for issuing the command associated with that user interface. For example, a mode is started when you select **File->Print**. This mode posts the **Print** dialog box and issues the print command when the user clicks **OK**.

Modules and toolsets

Modules and toolsets group functionality together. A GUI module is a grouping of similar functionality, such as the Part module in Abaqus/CAE. A GUI toolset is similar to a GUI module in that it groups similar functionality, but it generally contains more specific functionality that may be used by one or more GUI modules. The Datum tools in Abaqus/CAE are an example of a GUI toolset.

The application

The application is responsible for high-level activities, such as managing the GUI process used by the application and updating the state of the widgets. In addition, the application is responsible for interacting with the desktop's window manager.

Plug-ins and customized applications

There are two ways you can make use of the Abaqus GUI Toolkit—through the use of the plug-in architecture or by creating a custom application.

The Plug-in toolset is layered on top of Abaqus/CAE. First, the Abaqus/CAE application is built, and then the Plug-in toolset searches specific directories for files that add items into the top level **Plug-ins** menu. The Plug-in toolset will probably meet your needs if you intend only to add functionality to the standard Abaqus/CAE application, and it is sufficient to provide access to this functionality through the **Plug-ins** menu in the main menu bar. Plug-ins are described in detail in *Using plug-ins*.

In contrast, to create a customized application, you build the application from the ground up. You should write a customized application if, in addition to adding functionality to Abaqus/CAE, you want to modify some standard features of Abaqus/CAE. While creating a custom application offers the most flexibility, it requires more work than using the Plug-in toolset. However, a custom application allows you to modify aspects of an application that you cannot control using the Plug-in toolset. Specifically, a custom application allows you to do the following:

- Remove Abaqus/CAE modules or toolsets. When you create a custom application, you determine which modules and toolsets are loaded into the application and the order in which they appear.
- Modify Abaqus/CAE modules or toolsets. If you want to add or remove functionality from an Abaqus/CAE module, you must derive your module from an Abaqus/CAE module and then register your module instead of the Abaqus/CAE module. You follow a similar procedure if you want to add or remove functionality from an Abaqus/CAE toolset.
- Change the application name and version numbers. When you create a custom application, you create a startup script that initializes the application object with the name of your application and its version numbers.
- Control the startup command and license token used. When you create a custom application, you modify the site configuration file that defines the command used to start the application. You also modify the same site configuration file to specify the license token that is checked out when the application starts.

Running the prototype application

A custom application called the prototype application is available from the SIMULIA Community.
The prototype application allows you to experiment with the contents of your dialog box until you produce a design that is satisfactory. You can start the prototype application, make changes to the code that controls the contents of the dialog box, and instantly see those changes reflected in the application.

The SIMULIA Community (*SIMUILA Community > Learning Resources > 3DEXPERIENCE and Traditional Products > Abaqus > Plug-ins/Scripts*) provides examples of plug-ins and customized applications as well as access to a community of users that fosters the advance of the Abaqus Scripting Interface and the Abaqus GUI Toolkit. Search in this community for "Prototype Example" to download the zip file for the prototype application, then unzip the files and go to the directory containing the files that you downloaded. To use the prototype application, open the file `testDB.py` in a text editor. From the system prompt, enter the following:

```
abaqus cae -custom prototypeApp -noStartup
```

The `-custom` parameter indicates that you are executing an application that is a customized version of Abaqus/CAE. The `-noStartup` parameter indicates that you want to start Abaqus/CAE without displaying the startup screen. For more information, see *Abaqus/CAE Execution*.

The application creates two icons in the toolbox, as shown in *Figure 1*.

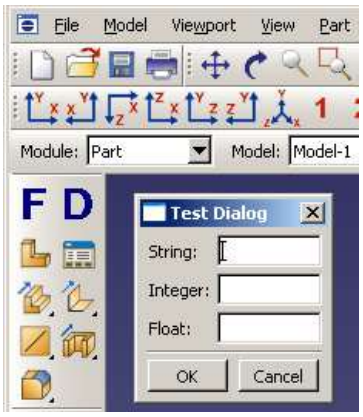


Figure 1: The prototype application.

The F icon reloads the form code (`testForm.py`); the D icon reloads the dialog box code (`testDB.py`). If you make changes to the form code, click on the F icon to reload that file. If you make changes to the dialog box code, click on the D icon to reload that file. You do not need to exit and restart Abaqus/CAE to see your changes in the form or dialog box.

For example, try the following:

- Click the D icon to post the dialog box and note the text labels shown in the dialog box.
- Click **Cancel** in the dialog box to unpost it.
- Change one of the labels in `testDB.py` and save the file.
- Click the D icon to post the dialog box again. You will see the modified label in the dialog box.

When you click **OK** in the dialog box, the kernel command issued by the dialog box is written to the message area, rather than being executed by Abaqus/CAE. This allows you to debug the command before you try to execute it in the kernel.

After you have debugged your form and dialog box code, you can modify the form to issue the command to the kernel by following the example in *Form example*. You can connect the form to your GUI, instead of to the  icon, by following the example shown in *Examining a GUI module example*.

Building Dialog Boxes

This part describes the components of a dialog box and how you create the components using the Abaqus GUI Toolkit.

In this section:

- *Widgets*
- *Layout managers*
- *Dialog boxes*

Widgets

This section describes how you can create widgets in your application.
There are many widgets in the Abaqus GUI Toolkit; however, only the most commonly used widgets are described here. You should refer to the *Abaqus GUI Toolkit Reference Guide* for a complete listing of widget classes.

In this section:

- *Labels and buttons*
- *Text widgets*
- *Lists and combo boxes*
- *Range widgets*
- *Tree widgets*
- *Table widget*
- *Miscellaneous widgets*
- *The create method*
- *Widgets and fonts*

Labels and buttons

This section describes the widgets in the Abaqus GUI Toolkit that use labels and buttons.

In this section:

- *About labels and buttons*
- *Labels*
- *Push buttons*
- *Check buttons*
- *Radio buttons*
- *Menu buttons*
- *Popup menus*
- *Toolbar and toolbox buttons*
- *Flyout buttons*
- *Color buttons*

About labels and buttons

Several widgets in the Abaqus GUI Toolkit support labels.
If you want to put a label before a text field, for example, you should use AFXTextField instead of creating a horizontal frame and adding a label widget and a text field widget. The following sections describe the specific widgets that support labels.

The label and button constructors all take a text string argument. This text string can consist of three parts, where each part is separated by `\t`. The three parts of the text string are

Text

The text displayed by the widget.

Tip text

The text displayed when the cursor is held over the widget for a short period of time. If there is only an icon associated with a widget, you must supply the tip text.

Help text

The text displayed in the application's status bar, assuming that the application has a status bar.

In addition, you can define a keyboard accelerator for the widget by preceding one of the characters in the text with an ampersand (`&`) character. For example, if you specify the string `&Calculate` for a button, the button label will appear as shown in *Figure 1*. You can use the accelerator to invoke the button by pressing the **[Alt]** key along with the **[C]** key.

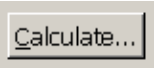


Figure 1: A keyboard accelerator applied to a button.

Labels

The FXLabel widget displays a read-only string. FXLabel can also display an optional icon.

```
FXLabel(parent, 'This is an FXLabel.\tThis is\nthe tooltip')
```



Figure 1: An example of a text label from FXLabel.

Push buttons

The FXButton widget contains a label and/or an icon.
When the user clicks the button, an immediate action is invoked.

```
FXButton(parent, 'This is an FXButton')
```

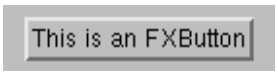


Figure 1: An example of a button from FXButton.

Check buttons

The FXCheckButton widget provides an “On/Off” toggling capability. The button also supports a third “Maybe” or “Some” state.
The “Maybe” state is often used to represent a partial selection; for example, the AFXOptionTreeList widget makes use of the “Maybe” state. You can set the “Maybe” state only programmatically; the user cannot toggle the button to this state.

```
FXCheckButton(parent, 'This is an FXCheckButton')
```

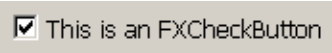


Figure 1: An example of a check button and a label from FXCheckButton.

Radio buttons

The FXRadioButton widget provides a one-of-many selection from a group of buttons.

```
FXRadioButton(parent, 'This is FXRadioButton 1')
FXRadioButton(parent, 'This is FXRadioButton 2')
FXRadioButton(parent, 'This is FXRadioButton 3')
```

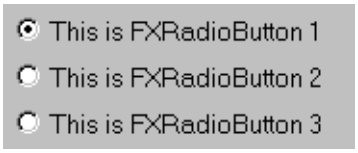


Figure 1: An example of radio buttons from FXRadioButton.

Menu buttons

A menu consists of a menu title created by AFXMenuTitle, a menu pane created by AFXMenuPane, and a menu command created by AFXMenuCommand.

The menu title resides in a menu bar and controls the display of the menu pane associated with the menu title. The menu pane contains menu commands. Menu commands are buttons that generally invoke some action. A menu pane can also contain a cascading menu created by AFXMenuCascade. A cascading menu provides submenus within a menu. *Figure 1* illustrates the components of a menu.

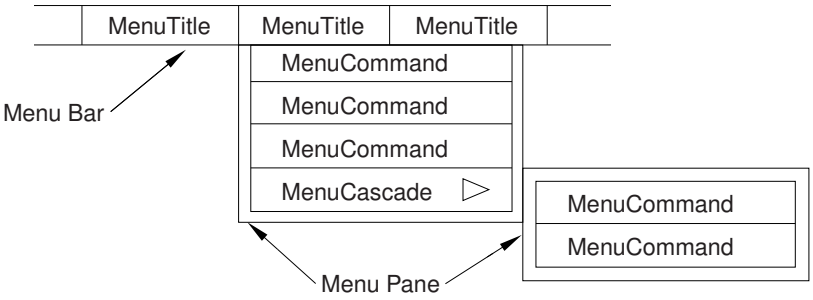


Figure 1: The components of a menu.

The following example illustrates the use of cascading menus:

```
menu = AFXMenuPane(self)
AFXMenuTitle(self, '&Menu1', None, menu)
AFXMenuCommand(self, menu, '&Item 1', None, form1,
    AFXMode.ID_ACTIVATE)
subMenu = AFXMenuPane(self)
AFXMenuCascade(self, menu, '&Submenu', None, subMenu)
AFXMenuCommand(self, subMenu, '&Subitem 1', None,
    form2, AFXMode.ID_ACTIVATE)
```

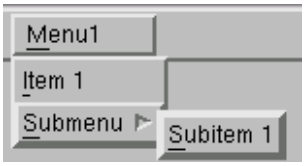


Figure 2: An example of cascading menu buttons from `AFXMenuCascade`.

In addition to specifying a mnemonic using the `&` syntax described in *Labels and buttons*, you can specify an accelerator in the menu item's label. You specify an accelerator by separating it from the button's text by a `\t`. For example,

```
AFXMenuCommand(self, menu, 'Graphics Options...\tCtrl+G', None,
    GraphicsOptionsForm(self), AFXMode.ID_ACTIVATE)
```

Popup menus

You can create a popup menu that appears when the user clicks mouse button 3 over a widget.

The following statements illustrate how you can create a popup menu that contains two buttons that appear when the user clicks mouse button 3 over a tree widget:

```
# In the dialog box constructor:

    def __init__(self, form):

        ...

        FXMAPFUNC(self, SEL_RIGHTBUTTONPRESS, self.ID_TREE,
            MyDB.onCmdPopup)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_TEST1,
            MyDB.onCmdTest1)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_TEST2,
            MyDB.onCmdTest2)

        self.menuPane = None

        FXTreeList(self, 5, self, self.ID_TREE,
            LAYOUT_FILL_X|LAYOUT_FILL_Y|
            TREELIST_SHOWS_BOXES|TREELIST_SHOWS_LINES|
            TREELIST_ROOT_BOXES|TREELIST_BROWSESELECT)
        ...

    def onCmdPopup(self, sender, sel, ptr):

        if not self.menuPane:
            self.menuPane = FXMenuPane(self)
            FXMenuCommand(self.menuPane, 'Test1', None, self,
                self.ID_TEST1)
            FXMenuCommand(self.menuPane, 'Test2', None, self,
                self.ID_TEST2)
            self.menuPane.create()

        status, x, y, buttons = self.getCursorPosition()
        x, y = self.translateCoordinatesTo(self.getRoot(), x, y )
        self.menuPane.popup (None, x, y)

        return 1
```



Note: The AFXTable has its own popup menu commands that you should use in place of the approach described in this section.

Toolbar and toolbox buttons

The AFXToolButton widget displays no text in its button, but the button generally has a tool tip.

You group the buttons created by AFXToolButton into toolbars using AFXToolbarGroups or into toolboxes using AFXToolboxGroups. AFXToolbarGroups and AFXToolboxGroups provide visual grouping between buttons in the toolbar or toolbox. For example,

```
# Create toolbar icons
#
group = AFXToolbarGroup(self)
AFXToolButton(group, '\tMy Module\nToolbar Button',
    icon, sel)

# Create toolbox icons
#
group = AFXToolboxGroup(self)
AFXToolButton(group, '\tMy Module\nToolbox Button',
    icon, sel)
```

Flyout buttons

The AFXFlyoutButton widget displays a flyout popup window.

The flyout popup window contains AFXFlyoutItem widgets and appears when the user presses mouse button 1 on the button and holds down mouse button 1 for a certain time span. If the user simply clicks mouse button 1 quickly on the button, the flyout popup window will not be displayed, and the flyout button will act just like a regular button. The AFXFlyoutButton widget displays the icon of the current target along with a right triangle in the lower right corner to indicate that a flyout popup window can be invoked. For example,

```
group = AFXToolbarGroup(self)
popup = FXPopup(getAFXApp().getAFXMainWindow())
AFXFlyoutItem(popup, '\tFlyout Button 1', squareIcon)
AFXFlyoutItem(popup, '\tFlyout Button 2', circleIcon)
AFXFlyoutItem(popup, '\tFlyout Button 3', triangleIcon)
AFXFlyoutButton(group, popup)
popup.create()
```

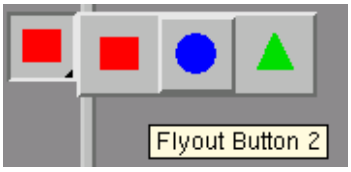


Figure 1: An example of flyout buttons from `AFXFlyoutItem`.

Color buttons

The AFXColorButton widget displays a push button that shows a color.

Clicking the button posts the color selection dialog box, which the user can use to change the value of the color for the button. For example,

```
AFXColorButton(parent, 'Color:')
```



Figure 1: An example of an `AFXColorButton`.

When connected to an AFXStringKeyword, this widget will assign the value of the button's current color to the keyword in hex format; for example, "#FF0000".

Text widgets

This section describes the widgets in the Abaqus GUI Toolkit that allow the user to input text.

In this section:

- *Single line text field widget*
- *Multi-line text widget*

Single line text field widget

The AFXTextField widget provides a single line text entry field.

AFXTextField extends the capability of the standard FXTextField widget with the following:

- An optional label.
- Support for a toggled version and a read-only state.
- An additional numeric type (complex).
- Horizontal and vertical layouts.

For example,

```
AFXTextField(parent, 8, 'String AFXTextField')
```



Figure 1: An example of a single-line text field from `AFXTextField`.

Text fields are generally connected to keywords, and the type of the keyword determines the type of input allowed in the text field. For example, if the text field is connected to an integer keyword, the keyword will verify that the input in the text field is a valid integer. You do not need to specifiy any option flags for the text field to get this behavior. Complex text fields are an exception to this—to display the extra field needed to collect complex input, you must specify the bit flag shown in the following example:

```
AFXTextField(parent, 8, 'Complex AFXTextField',
    None, 0, AFXTEXTFIELD_COMPLEX)
```



Figure 2: An example of a single-line complex numeric field from `AFXTextField`.

Toggled variation

In many cases a check button precedes a labeled text field. The check button allows the user to toggle the component on or off; when the component is toggled off, the text field becomes disabled. The AFXTextField widget creates a check button with this behavior when you supply the AFXTEXTFIELD_CHECKBUTTON flag. The following example creates a check button with a text field. It also configures the widget in a vertical orientation so that the label is above the text field.

```
AFXTextField(parent, 8, 'AFXTextField', None, 0,
    AFXTEXTFIELD_CHECKBUTTON|AFXTEXTFIELD_VERTICAL)
```

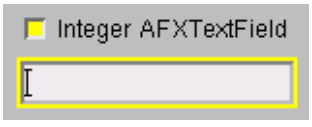


Figure 3: An example of a check button with a labeled text field from `AFXTextField`.

Non-editable variation

In some cases you may want to change the behavior of a text field so that it cannot be edited by the user; for example, when a particular check button in the dialog box is not set. In this case, you can make the text field

non-editable when the check button is unset by calling the `setEditable(False)` method of the AFXTextField widget.

Read-only variation

In some cases you may want to change the behavior of a text field so that it cannot be edited by the user and appears as a label, making it clear that the user cannot change its contents. For example, when you are using the Load module in Abaqus/CAE, there are some values that you can specify in the analysis step in which the load was created but you cannot change in subsequent steps. The AFXTextField widget supports a read-only state through the `setReadOnlyState` method. For example,

```
tf = AFXTextField(parent, 8,
    'String AFXTextField in read-only mode:', keyword)
tf.setReadOnlyState(True)
```

Multi-line text widget

FXText provides a multi-line text entry area.

For example,

```
text = FXText(parent, None, 0,
    LAYOUT_FIX_WIDTH|LAYOUT_FIX_HEIGHT, 0, 0, 300, 100)
text.setText('This is an FXText widget')
```

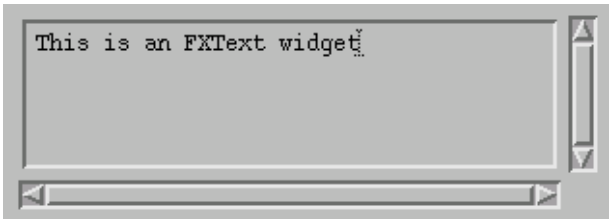


Figure 1: An example of a multi-line text entry area from `FXText`.

Lists and combo boxes

This section describes the widgets in the Abaqus GUI Toolkit that allow you to choose one or more items from a list.

- You use a list widget when there is enough room in the GUI and when it is helpful to display all or most of the choices at the same time.
- You use a combo box to conserve space in the GUI and when it is preferable to display only the current choice.

In this section:

- *Lists*
- *Combo boxes*
- *List boxes*

Lists

AFXList allows one or more selections from its items.

The list created by AFXList supports the following selection policies:

LIST_SINGLESELECT

The user can select zero or one items.

LIST_BROWSESELECT

One item is always selected.

LIST_MULTIPLESELECT

The user can select zero or more items.

LIST_EXTENDEDSELECT

The user can select zero or more items; drag-, shift-, and control-selections are allowed.

The AFXDialog base class has special code designed to handle double-click messages from a list. If the user double-clicks in a list, the dialog box first attempts to call the **Apply** button message handler. If the **Apply** button message handler is not found, the dialog attempts to call the **Continue** button message handler. If the **Continue** button message handler is not found, the dialog attempts to call the **OK** button message handler. As a result, you do not need to do anything in your script to get this behavior.

However, if you have special double-click processing needs, you can turn off this double-click behavior by specifying AFXLIST_NO_AUTOCOMMIT as one of the list's option flags. If you turn off the double-click behavior, you must catch the SEL_DOUBLECLICKED message from the list in your dialog box and handle it appropriately.



Note:

Because the list may be used in combination with other types of widgets, the list does not draw a border around itself. As a result, if you want a border around the list, you must provide the border by placing the list inside some other widget, such as a frame. If you do not want a horizontal scrollbar, use the HSCROLLING_OFF flag; this flag forces the list to size its width to fit its widest item.

The following is an example of a list within a vertical frame:

```
vf = FXVerticalFrame(parent, FRAME_THICK|FRAME_SUNKEN,
    0, 0, 0, 0, 0, 0, 0, 0)
list = AFXList(vf, 3, tgt, sel, LIST_BROWSESELECT|HSCROLLING_OFF)
list.appendItem('Thin')
list.appendItem('Medium')
list.appendItem('Thick')
```



Figure 1: An example of a list with a frame from `AFXList`.

Combo boxes

AFXComboBox provides a one-of-many selection from its items.
AFXComboBox combines a read-only text field with a drop-down list.

After the parent argument, the next three arguments to the AFXComboBox constructor are the width of the text field, the number of visible list items when the list is exposed, and the label. If you specify the width as zero, the combo box will automatically size itself to the widest item in its list. For example,

```
comboBox = AFXComboBox(p, 0, 3, 'AFXComboBox:')
comboBox.appendItem('Item 1')
comboBox.appendItem('Item 2')
comboBox.appendItem('Item 3')
```

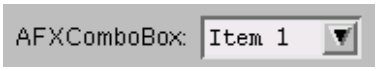


Figure 1: An example of a combo box from AFXComboBox.

List boxes

The AFXListBox widget provides a one-of-many selection from its items.

AFXListBox differs from AFXComboBox in that the items displayed by AFXListBox can include icons. For example,

```
listBox = AFXListBox(parent, 3, 'AFXListBox:', keyword)
listBox.appendItem('Item 1', thinIcon)
listBox.appendItem('Item 2', mediumIcon)
listBox.appendItem('Item 3', thickIcon)
```

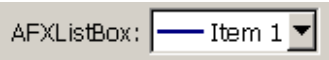


Figure 1: An example of a list box from `AFXListBox`.

Range widgets

This section describes the widgets in the Abaqus GUI Toolkit that allow the user to specify a value within certain bounds.

In this section:

- *Sliders*
- *Spinners*

Sliders

The AFXSlider widget provides a handle that the user can drag to set a value using only the mouse.

AFXSlider extends the capability of the FXSlider widget by providing the following:

- An optional title.
- Minimum and maximum range labels.
- The ability to display the current value above the drag handle.

For example,

```
slider = AFXSlider(p, None, 0,
    AFXSLIDER_INSIDE_BAR|AFXSLIDER_SHOW_VALUE|LAYOUT_FILL_X)
slider.setMinLabelText('Min')
slider.setMaxLabelText('Max')
slider.setDecimalPlaces(1)
slider.setRange(20, 80)
slider.setValue(50)
```

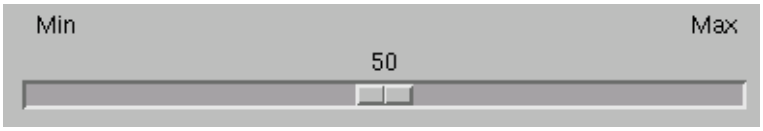


Figure 1: An example of a slider from `AFXSlider`.

Spinners

The AFXSpinner widget combines a text field and two arrow buttons.

The arrows increment the integer value shown in the text field. AFXSpinner extends the capability of the FXSpinner widget by providing an optional label. For example,

```
spinner = AFXSpinner(p, 4, 'AFXSpinner:')
spinner.setRange(20, 80)
spinner.setValue(50)
```



Figure 1: An example of a spinner from `AFXSpinner`.

The AFXFloatSpinner widget is similar to the AFXSpinner widget, but it allows floating point values.

Tree widgets

This section describes the tree widgets in the Abaqus GUI Toolkit.
A tree widget arranges its children in a hierarchical fashion and allows the various branches to be expanded or collapsed. A file browser such as Microsoft Windows Explorer is a common example of an application that makes use of a tree widget.

In this section:

- *Tree list*
- *Option tree list*

Tree list

The FXTreeList widget provides a tree structure of children that can be expanded and collapsed.

The `FXTreeList` constructor is defined by the following prototype:

```
FXTreeList(p, nvis, tgt=None, sel=0,
    opts=TREELIST_NORMAL, x=0, y=0, w=0, h=0)
```

The arguments to the `FXTreeList` constructor are described in the following list:

parent

The first argument in the constructor is the parent. An `FXTreeList` does not draw a frame around itself; therefore, you may want to create an `FXVerticalFrame` to use as the parent of the tree. You should zero out the padding in the frame so that the frame wraps tightly around the tree.

number of visible items

The number of items that will be visible when the tree is first displayed.

target and selector

You can specify a target and selector in the tree constructor arguments.

opts

The option flags that you can specify in the tree constructor are shown in the following table:

Option flag	Effect
TREELIST_NORMAL (default)	TREELIST_EXTENDEDSELECT
TREELIST_EXTENDEDSELECT	Extended selection mode allows the user to drag-select ranges of items.
TREELIST_SINGLESELECT	Single selection mode allows the user to select up to one item.
TREELIST_BROWSESELECT	Browse selection mode enforces one single item to be selected at all times.
TREELIST_MULTIPLESELECT	Multiple selection mode is used for selection of individual items.
TREELIST_AUTOSELECT	Automatically select under cursor.
TREELIST_SHOWS_LINES	Show lines between items.
TREELIST_SHOWS_BOXES	Show boxes when item can expand.
TREELIST_ROOT_BOXES	Show root item boxes also.

The following statements show an example of creating a tree:

```
vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
tree = FXTreeList(vf, 5, None, 0,
    LAYOUT_FILL_X|LAYOUT_FILL_Y|
    TREELIST_SHOWS_BOXES|TREELIST_ROOT_BOXES|
    TREELIST_SHOWS_LINES|TREELIST_BROWSESELECT)
```

You add an item to a tree by supplying a parent and a text label. You begin by adding root items to the tree. Root items have a parent of `None`. The Abaqus GUI Toolkit provides several ways of adding items to a tree; however, the most convenient approach uses the `addItemLast` method, as shown in the following example:

```
vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
self.tree = FXTreeList(vf, 5, None, 0,
    LAYOUT_FILL_X|LAYOUT_FILL_Y|
    TREELIST_SHOWS_BOXES|TREELIST_ROOT_BOXES|
    TREELIST_SHOWS_LINES|TREELIST_BROWSESELECT)
option1 = self.tree.addItemLast(None, 'Option 1')
self.tree.addItemLast(option1, 'Option 1a')
self.tree.addItemLast(option1, 'Option 1b')
option2 = self.tree.addItemLast(None, 'Option 2')
self.tree.addItemLast(option2, 'Option 2a')
option2b = self.tree.addItemLast(option2, 'Option 2b')
self.tree.addItemLast(option2b, 'Option 2bi')
option3 = self.tree.addItemLast(None, 'Option 3')
```

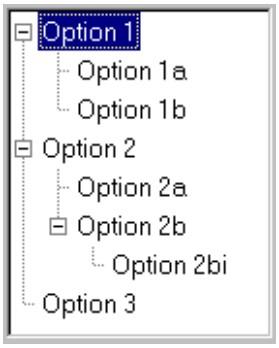


Figure 1: An example of a tree widget.

You can also specify icons to be used for each tree item. The “open” icon is displayed next to an item when it is selected; the “closed” icon is displayed when the item is not selected. These icons are not associated with the expanded/collapsed state of a branch. For example, Microsoft's Windows Explorer uses open and closed folder icons to show the selected state.

You can check if an item is selected using the tree's `isItemSelected` method. The tree widget will send its target a SEL_COMMAND message whenever the user clicks on an item. You can handle this message and then traverse all the items in the tree to find the selected item. The following example uses a message handler that assumes that the tree is browse-select and allows the user to select only one item at a time:

```
def onCmdTree(self, sender, sel, ptr):

    w = self.tree.getFirstItem()
    while(w):
        if self.tree.isItemSelected(w):
            # Do something here based on
            # the selected item, w.
            break
        if w.getFirst():
            w=w.getFirst()
            continue
        while not w.getNext() and w.getParent():
             w=w.getParent()
        w=w.getNext()
```

Option tree list

The AFXOptionTreeList widget provides a tree structure of children that can be toggled.

The tree structure includes branches along with leaves at the end of a branch. The user can toggle the leaves of the tree on or off. The user can also toggle the entire branch on or off. The toggle controls the settings of all the children of the branch—if the branch is toggled off, all the children are toggled off and vice versa. For example,

```
tree = AFXOptionTreeList(parent, 6)
tree.addItemLast('Item 1')
item = tree.addItemLast('Item 2')
item.addItemLast('Item 3')
item.addItemLast('Item 4')
item.addItemLast('Item 5')
```

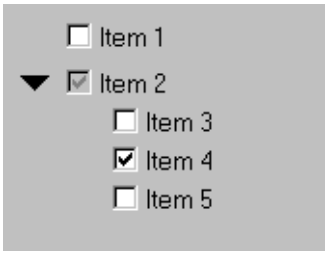


Figure 1: An example of an option tree list from AFXOptionTreeList.

Table widget

The `AFXTable` widget arranges items in rows and columns, similar to a spreadsheet.

In this section:

- *Table widget layout*
- *Table constructor*
- *Rows and columns*
- *Spanning*
- *Justification*
- *Editing*
- *Types*
- *List type*
- *Boolean type*
- *Icon type*
- *Color type*
- *Popup menu*
- *Colors*
- *Sorting*

Table widget layout

The `AFXTable` widget lays out a table that can have leading rows and columns, which serve as headings.

Figure 1 shows an example of how the Abaqus GUI Toolkit lays out a table.

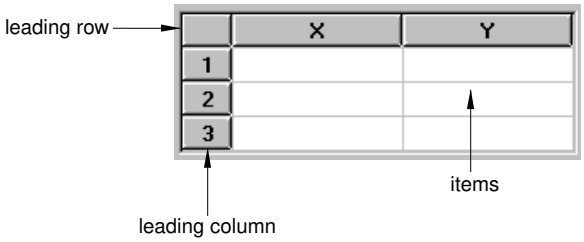


Figure 1: The layout of a table.

The `AFXTable` widget has many options and methods that allow a lot of flexibility when you are trying to configure a table for specific purposes. These options and methods are discussed in the following sections.

Table constructor

You can specify the parent, the number of visible rows and columns, the number of rows and columns, the target and selector, and the option flags in the `AFXTable` constructor.

The `AFXTable` constructor is defined by the following prototype:

```
AFXTable(p, numVisRows, numVisColumns, numRows, numColumns,
    tgt=None, sel=0, opts=AFXTABLE_NORMAL,
    x=0, y=0, w=0, h=0,
    pl= DEFAULT_MARGIN, pr=DEFAULT_MARGIN,
    pt=DEFAULT_MARGIN, pb=DEFAULT_MARGIN)
```

The `AFXTable` constructor has the following arguments:

parent

The first argument in the constructor is the parent. An `AFXTable` does not draw a frame around itself; therefore, you may want to create an `FXVerticalFrame` to use as the parent of the table. You should zero out the padding in the frame so that the frame wraps tightly around the table.

number of visible rows and columns

The number of rows and columns that will be visible when the table is first displayed. If the number of visible rows or columns is less than the total number of rows or columns in the table, the appropriate scroll bars are displayed.

number of rows and columns

The number of rows and columns to be created when the table is created. These numbers include leading rows and columns. If the size of the table is fixed, you specify the total number of rows and columns. If the size of the table is dynamic, you specify `1` row and `1` column (plus any leading rows or columns) and allow the user to add rows or columns as necessary.

target and selector

You can specify a target and selector in the table constructor arguments. A table is generally connected to an `AFXTableKeyword` with a selector of 0, unless the table has columns that are not directly related to the data required by the command to be sent to the kernel. If the table has columns that are not required by the kernel, you can specify the dialog as the target so that the data in the table can be processed appropriately by your code. You can also use the AFXColumnItems object to manage selection in particular table column automatically (for more information, see *Table keyword example*).

opts

The option flags that you can specify in the table constructor are shown in the following table:

Option flag	Effect
AFXTABLE_NORMAL (default)	AFXTABLE_COLUMN_RESIZABLE | LAYOUT_FILL_X | LAYOUT_FILL_Y
AFXTABLE_COLUMN_RESIZABLE	Allows columns to be resized by the user.
AFXTABLE_ROW_RESIZABLE	Allows rows to be resized by the user.
AFXTABLE_RESIZE	AFXTABLE_COLUMN_RESIZABLE | AFXTABLE_ROW_RESIZABLE

Option flag	Effect
AFXTABLE_NO_COLUMN_SELECT	Disallows selecting the entire column when its heading is clicked.
AFXTABLE_NO_ROW_SELECT	Disallows selecting the entire row when its heading is clicked.
AFXTABLE_SINGLE_SELECT	Allows up to one item to be selected.
AFXTABLE_BROWSE_SELECT	Enforces one single item to be selected at all times.
AFXTABLE_ROW_MODE	Selecting an item in a row selects the entire row.
AFXTABLE_EDITABLE	Allows all items in the table to be edited.

By default, the user can select multiple items in a table. To change this behavior, you should use the appropriate flag to specify either single select mode or browse select mode. In addition, you can specify whether the entire row should be selected when the user selects any item in the row. Abaqus/CAE exhibits this behavior in manager dialogs that contain more than one column.

The following statements creates a table with default settings:

```
# Tables do not draw a frame around their border.
# Therefore, add a frame widget with zero padding.

vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 2, 4, 2)
```



Figure 1: A table created with default settings.

Rows and columns

The table supports leading rows and columns.
Leading rows and columns are displayed as buttons using a bold text. Leading rows are displayed at the top of the table, and leading columns are displayed on the left side of the table.

The number of rows and columns that you specify in the table constructor are the total number of rows and columns, including the leading rows and columns. By default, the table has no leading rows or columns—you must set the leading rows and columns after the table is constructed using the appropriate table methods. You can also specify the labels to be displayed in these rows and columns. If you do not specify any labels for a leading row or column, it will be numbered automatically. You can set more than one label at once in a heading by using "\t" to separate the labels in a single string.

By default, no gridlines are drawn around items. You can control the visibility of the horizontal and vertical gridlines individually by using the following table methods:

- `showHorizontalGrid(True|False)`
- `showVerticalGrid(True|False)`

By default, the height of the rows is determined by the font being used for the table. The default width of a column is 100 pixels. You can override these values using the following table methods:

- `setRowHeight( row, height) # Height is in pixels`
- `setColumnWidth(column, width) # Width is in pixels`

The following example illustrates the use of some of these methods:

```
vf = FXVerticalFrame(parent, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0,0,0,0,0)
table = AFXTable(vf, 4, 3, 4, 3)
table.setLeadingColumns(1)
table.setLeadingRows(1)
table.setLeadingRowLabels('X\tY')
table.showHorizontalGrid(True)
table.showVerticalGrid(True)
table.setColumnWidth(0, 30)
```

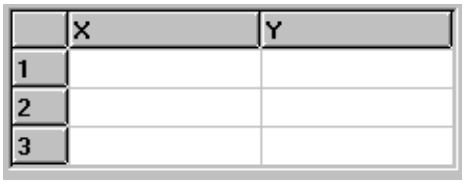


Figure 1: Leading rows and columns.

Spanning

You can make an item in a header row or column span more than one row or column.

```
vf = FXVerticalFrame(parent, FRAME_SUNKEN|FRAME_THICK,
     0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 3, 4, 3)
table.setLeadingColumns(1)
table.setLeadingRows(2)

# Corner item
table.setItemSpan(0, 0, 2, 1)

# Span top row item over 2 columns
table.setItemSpan(0, 1, 1, 2)
table.setLeadingRowLabels('Coordinates')
table.setLeadingRowLabels('X\tY', 1)

table.showHorizontalGrid(True)
table.showVerticalGrid(True)

table.setColumnWidth(0, 30)
```

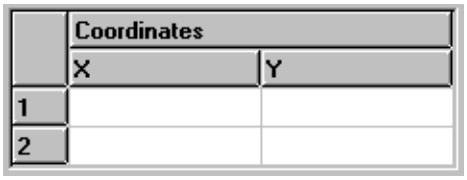


Figure 1: An example of spanning two header columns.

Justification

By default, the table displays entries left justified.

You can change how items are justified by using the following table methods:

- `setColumnJustify(column, justify)`
- `setItemJustify(row, column, justify)`

If you supply a value of `-1` for the column number, the `setColumn*` methods apply the setting to all columns in the table.

The following table shows the possible values for the justify argument:

Option flag	Effect
AFXTable .LEFT	Align items to the left side of the cell.
AFXTable .CENTER	Center items horizontally.
AFXTable .RIGHT	Align items to the right side of the cell.
AFXTable .TOP	Align items to the top of the cell.
AFXTable .MIDDLE	Center items vertically.
AFXTable .BOTTOM	Align items to the bottom of the cell.

The following example shows how you can change the justification:

```
vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 3, 4, 3)
table.setLeadingColumns(1)
table.setLeadingRows(1)
table.setLeadingRowLabels('X\tY')
table.showHorizontalGrid(True)
table.showVerticalGrid(True)
table.setColumnWidth(0, 30)

# Center all columns
table.setColumnJustify(-1, AFXTable.CENTER)
```

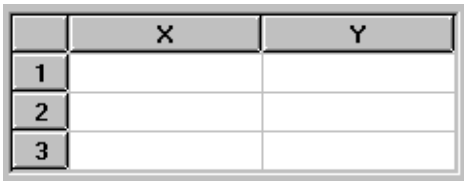


Figure 1: Justified column headings.

Editing

By default, no items in a table are editable. To make all items in a table editable, you must specify AFXTABLE_EDITABLE in the table constructor.

To change the editability of some items in a table, you can use the following table methods:

- `setColumnEditable(column, True|False)`
- `setItemEditable(row, column, True|False)`

Types

By default, all items in a table are text items. However, the table widget also supports the BOOL, COLOR, FLOAT, ICON, INT, LIST, and TEXT items.

The table widget also support the other types of items shown in the following table:

Type	Effect
BOOL	Item shows an icon, clicking on it toggles between a true and false icon.
COLOR	Item shows a color button
FLOAT	Item shows text, a text field is used to edit the value
ICON	Item shows an icon, it is not editable.
INT	Item shows text, a text field is used to edit the value
LIST	Item shows text, a combo box is used to edit the value.
TEXT	Item shows text, a text field is used to edit the value.

You can change the type of a column or the type of an individual item using the following table methods:

- `setColumnType(column, type)`
- `setItemType(row, column, type)`

Setting the type to FLOAT or INT does not affect data entry to the table; the user may enter anything into these types of items (this allows for expression evaluation). However, when using the table's `getItemIntValue` or `getItemFloatValue` methods you should be sure that the type of the item that you are reading is INT or FLOAT, respectively, or the wrong value may be returned. In general, you should make use of the AFXTableKeyword and set the column types so that the table's values are automatically evaluated correctly.

List type

If you want to allow the user to specify a value in a column by selecting from a list of items, you must first set the column to be of type LIST.
You then create a list and assign it to that column. When the user clicks on an item in that column, the table will display a noneditable combo box that contains the entries from the list.

The following example illustrates how you can create a combo box within a table cell:

```
vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 2, 4, 2, None, 0,
    AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
table.setLeadingRows(1)
table.setLeadingRowLabels('Size\tQuantity')
table.showHorizontalGrid(True)
table.showVerticalGrid(True)

listId = table.addList('Small\tMedium\tLarge')
table.setColumnType(0, AFXTable.LIST)
table.setColumnListId(0, listId)
```

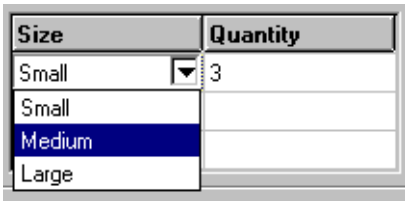


Figure 1: A combo box within a table cell.

You can also add list items that contain icons using the `appendListItem` method of the table.

```
icon = createGIFIcon('myIcon.gif')
table.appendListItem(listId, 'Extra large', icon)
```

When you connect a table keyword to a table that contains lists, you must set the column type of the table keyword appropriately. If the list contains only text, you can set the column type to AFXTABLE_TYPE_STRING, which sets the value of the keyword to the text of the item selected in the list. Similarly, if the list contains only icons, you can set the column type to AFXTABLE_TYPE_INT, which sets the value of the keyword to the index of the item selected in the list. If the list contains both text and icons, you can use either setting for the column type.

Boolean type

If you want to allow the user to specify a value in a table be either True or False, you must set the type of the column to be BOOL.

The value of a Boolean item is toggled each time you click the item. By default, a blank icon represents False and a check mark icon represents True. Checkbox style icons for an editable table are shown below. The following example illustrates how you can include Boolean items in a table:

```
vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 2, 4, 2, None, 0,
    AFXTABLE_NORMAL)
table.setLeadingRows(1)
table.setLeadingRowLabels('Nlgeom\tStep')
table.showHorizontalGrid(True)
table.showVerticalGrid(True)
table.setColumnType(0, table.BOOL)
table.setColumnWidth(0, 50)
table.setColumnJustify(0, AFXTable.CENTER)
```

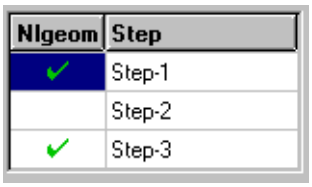


Figure 1: Boolean items in a table.

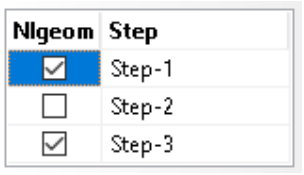


Figure 2: Boolean items in an editable table.

If you do not want to use the default icons, you can set your own true and false icons, as shown in the following example:

```
vf = FXVerticalFrame(gb, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 2, 4, 2, None, 0,
    AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
table.setLeadingRows(1)
table.setLeadingRowLabels('State\tLayer')
table.showHorizontalGrid(True)
table.showVerticalGrid(True)
table.setColumnType(0, table.BOOL)
table.setColumnWidth(0, 50)
table.setColumnJustify(0, AFXTable.CENTER)

from appIcons import lockedData, unlockedData
trueIcon = FXXPMIcon(getAFXApp(), lockedData)
falseIcon = FXXPMIcon(getAFXApp(), unlockedData)
table.setDefaultBoolIcons(trueIcon, falseIcon)
```

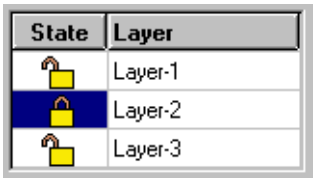


Figure 3: Defining your own true and false icons.

Icon type

If you want to display an icon in an item, you must set the type of the column to be ICON and assign the icons to be shown.

This type of column is not editable by the user. The following example shows how you can include an icon in a table cell:

```
vf = FXVerticalFrame(parent, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 2, 4, 2, None, 0,
    AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
table.setLeadingRows(1)
table.setLeadingRowLabels(' \tStatus')
table.showHorizontalGrid(True)
table.showVerticalGrid(True)
table.setColumnType(0, table.ICON)
table.setColumnWidth(0, 30)
table.setColumnJustify(0, AFXTable.CENTER)

from appIcons import circleData, squareData
circleIcon = FXXPMIcon(getAFXApp(), circleData)
squareIcon = FXXPMIcon(getAFXApp(), squareData)
table.setItemIcon(1, 0, circleIcon)
table.setItemIcon(2, 0, squareIcon)
table.setItemIcon(3, 0, circleIcon)
```

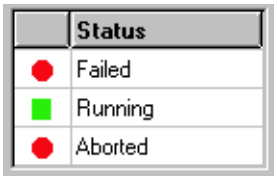


Figure 1: Including icons in table cells.

Color type

If you want to display a color button in a table, you must set the type to COLOR.

If the table is editable, the user can use the color button to change the color via the color selection dialog box. The color button is a flyout button that can have up to three flyout items, one for a specific color, one for a default color, and one for an “as-is” color. Refer to the Color Code dialog box in Abaqus/CAE to see examples of how these options may be used. The options are specified using the flags in the following table:

Option flag	Effect
COLOR_INCLUDE_COLOR_ONLY	Include only the color flyout item.
COLOR_INCLUDE_AS_IS	Include the “as–is” (=) flyout item.
COLOR_INCLUDE_DEFAULT	Include the default (*) flyout item.
COLOR_INCLUDE_ALL	Include all of the flyout items.

The following example shows how you can display color buttons in a table:

```
vf = FXVerticalFrame(
    gb, FRAME_SUNKEN|FRAME_THICK, 0,0,0,0, 0,0,0,0)
table = AFXTable(
    vf, 4, 2, 4, 2, None, 0, AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
table.setLeadingRows(1)
table.setLeadingRowLabels('Name\tColor')
table.setColumnType(1,AFXTable.COLOR)
table.setColumnColorOptions(
    1, AFXTable.COLOR_INCLUDE_COLOR_ONLY)
table.setItemText(1, 0, 'Part-1')
table.setItemText(2, 0, 'Part-2')
table.setItemText(3, 0, 'Part-3')
table.setItemColor(1,1, '#FF0000')
table.setItemColor(2,1, '#00FF00')
table.setItemColor(3,1, '#0000FF')
```

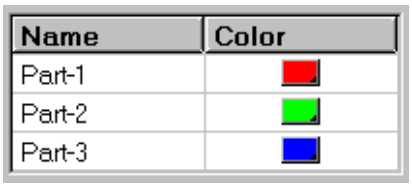


Figure 1: Including icons in table cells.

Popup menu

You can add a popup menu to the table by specifying the appropriate flags using the `setPopupOptions` method. The menu will be posted when the user clicks mouse button 3 anywhere over the table.

The following options are supported in the popup menu:

Option flag	Effect
POPUP_NONE (default)	No popup menu will be displayed.
POPUP_CUT	Adds a Cut button to the popup menu.
POPUP_COPY	Adds a Copy button to the popup menu.
POPUP_PASTE	Adds a Paste button to the popup menu.
POPUP_EDIT	POPUP_CUT | POPUP_COPY | POPUP_PASTE
POPUP_INSERT_ROW	Adds Insert Row Before/After buttons to the popup menu.
POPUP_INSERT_COLUMN	Adds Insert Column Before/After buttons to the popup menu.
POPUP_DELETE_ROW	Adds Delete Rows button to the popup menu.
POPUP_DELETE_COLUMN	Adds Delete Columns button to the popup menu.
POPUP_CLEAR_CONTENTS	Adds Clear Contents/Table buttons to the popup menu.
POPUP_MODIFY	POPUP_INSERT_ROW | POPUP_INSERT_COLUMN | POPUP_DELETE_ROW | POPUP_DELETE_COLUMN | POPUP_CLEAR_CONTENTS
POPUP_READ_FROM_FILE	Adds Read from File button to the popup menu. **Note:** Include POPUP_INSERT_ROW with POPUP_READ_FROM_FILE to allow automatic expansion of the table for data files with more lines than the current table definition.
POPUP_WRITE_TO_FILE	Adds Write to File button to the popup menu.
POPUP_ALL	POPUP_EDIT | POPUP_MODIFY | POPUP_READ

You can also add a custom button to the popup menu by using the table's `appendClientPopupItem` method, as shown in *Figure 1*.

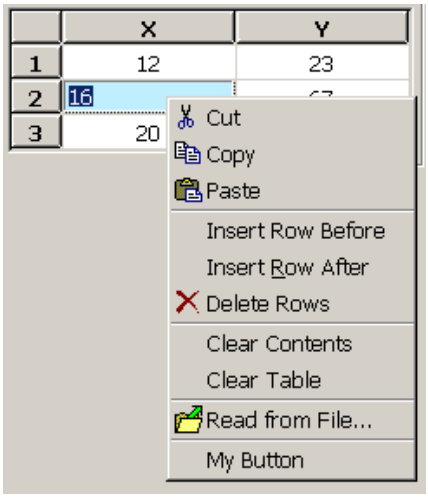


Figure 1: Popup menu options.

The following example shows how you can enable various popup menu options:

```
vf = FXVerticalFrame(parent, FRAME_SUNKEN|FRAME_THICK,
     0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 3, 4, 3, None, 0,
    AFXTABLE_NORMAL|AFXTABLE_EDITABLE)

table.setLeadingColumns(1)
table.setLeadingRows(1)
table.setLeadingRowLabels('X\tY')

table.showHorizontalGrid(True)
table.showVerticalGrid(True)

table.setColumnWidth(0, 30)

# Center all columns
table.setColumnJustify(-1, table.CENTER)

table.setPopupOptions(
    AFXTable.POPUP_CUT|AFXTable.POPUP_COPY
   |AFXTable.POPUP_PASTE
   |AFXTable.POPUP_INSERT_ROW
   |AFXTable.POPUP_DELETE_ROW
   |AFXTable.POPUP_CLEAR_CONTENTS
   |AFXTable.POPUP_READ_FROM_FILE
)
table.appendClientPopupItem('My Button', None, self,
    self.ID_MY_BUTTON)
FXMAPFUNC(self, SEL_COMMAND, self.ID_MY_BUTTON, MyDB.onCmdMyBtn)
```

Colors

Items in a table that display characters have two sets of colors—the normal color and the selected color. In addition, each item has a background color and a text color.

To change these colors, the table widget provides the following controls:

- Item text color
- Item background color
- Selected item text color
- Selected item background color
- Item color (color button) (The color button is described in *Color buttons*.)

You can control the text color of items that display characters using the `setItemTextColor` method. Items that display characters include strings, numbers, and lists. You can control the text color of these items when they are selected by using the `setSelTextColor` method. You can control the background color of any item by using the `setItemBackColor` method. You can control the background color of any item when it is selected by using the `setSelBackColor` method.

If you do not want the colors to change when the user selects an item, you can set the colors used for items that are selected to be the same as the colors used for items that are not selected. This approach is shown in the following example:

```
itemColor = table.getItemBackColor(1,1)
table.setSelBackColor(itemColor)
itemTextColor = table.getItemTextColor(1,1)
table.setSelTextColor(itemTextColor)
```

You can set colors using the color name or by specifying RGB values using the `FXRGB` function. (For a list of valid color names and their corresponding RGB values, see *Colors and RGB values*.) Both methods are illustrated in the following example:

```
vf = FXVerticalFrame(parent, FRAME_SUNKEN|FRAME_THICK,
    0,0,0,0, 0,0,0,0)
table = AFXTable(vf, 4, 2, 4, 2, None, 0,
    AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
table.setLeadingRows(1)
table.setLeadingRowLabels('Name\tDescription')

table.showHorizontalGrid(True)
table.showVerticalGrid(True)

table.setItemTextColor(1,0, 'Blue')
table.setItemTextColor(1,1, FXRGB(0, 0, 255))

table.setItemBackColor(3,0, 'Pink')
table.setItemBackColor(3,1, FXRGB(255, 192, 203))
```

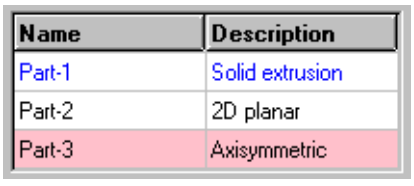


Figure 1: Setting colors for table items.

Sorting

You can set a column in a table to be sortable.

If a column is set to be sortable and the user clicks on its heading, a graphic will be displayed in the heading that shows the order of the sort. You must write the code that performs the actual sorting in the table—the table itself provides only the graphical feedback in the heading cell. For example:

```
class MyDB(AFXDataDialog):

    def __init(self):

        ...

        # Handle clicks in the table.
        FXMAPFUNC(self, SEL_CLICKED, self.ID_TABLE,
                      MyDB.onClickTable)
        ...

        # Create a table.
        vf = FXVerticalFrame(
            parent, FRAME_SUNKEN|FRAME_THICK, 0,0,0,0, 0,0,0,0)
        self.sortTable = AFXTable(vf, 4, 3, 4, 3, self,
            self.ID_TABLE, AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
        self.sortTable.setLeadingRows(1)
        self.sortTable.setLeadingRowLabels('Name\tX\tY')
        self.sortTable.setColumnSortable(1, True)
        self.sortTable.setColumnSortable(2, True)
        ...

    def onClickTable(self, sender, sel, ptr):

        status, x, y, buttons = self.sortTable.getCursorPosition()
        column = self.sortTable.getColumnAtX(x)
        row = self.sortTable.getRowAtY(y)

        # Ignore clicks on table headers.
        if row != 0 or column == 0:
            return

        values = []
        index = 1
        for row in range(1, self.sortTable.getNumRows()):
            values.append( (self.sortTable.getItemFloatValue(
                row, column), index) )
            index += 1

        values.sort()
        if self.sortTable.getColumnSortOrder(column) == \
            AFXTable.SORT_ASCENDING:
                values.reverse()

        items = []
        for value, index in values:
            name = self.sortTable.getItemValue(index, 0)
            xValue = self.sortTable.getItemValue(index, 1)
            yValue = self.sortTable.getItemValue(index, 2)
            items.append( (name, xValue, yValue) )

        row = 1
```

```
        for name, xValue, yValue in items:
            self.sortTable.setItemValue(row, 0, name)
            self.sortTable.setItemValue(row, 1, xValue)
            self.sortTable.setItemValue(row, 2, yValue)
            row += 1
```

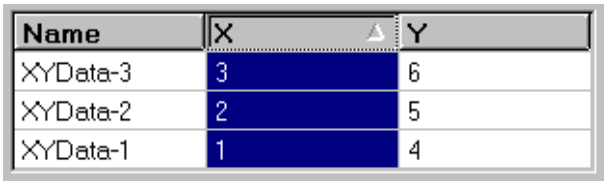


Figure 1: Sorting table items.

Miscellaneous widgets

This section describes a set of miscellaneous widgets in the Abaqus GUI Toolkit that you can use in your applications.

In this section:

- *Separators*
- *Notes and warnings*
- *Progress bar*

Separators

The FXHorizontalSeparator widget and the FXVerticalSeparator widget provide a visual separator to allow separating elements in a GUI. The Abaqus GUI Toolkit also includes a FXMenuSeparator widget that you can use to separate items in a menu pane.

For example,

```
FXLabel(parent, 'This is a label above an FXHorizontalSeparator')
FXHorizontalSeparator(parent)
FXLabel(parent, 'This is a label below an FXHorizontalSeparator')
```

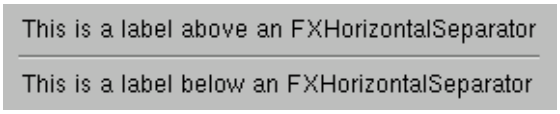


Figure 1: An example of a horizontal separator from `FXHorizontalSeparator`.

Notes and warnings

The AFXNote widget provides a convenient way to display notes or warnings in a dialog box.

AFXNote displays either the word “Note” or the word “Warning” in a bold font. AFXNote also aligns messages that contain more than one line. For example,

```
AFXNote(parent, 'This is an AFXNote information note\n'
    'that wraps on two lines.')
AFXNote(parent, 'This is an AFXNote warning note!', NOTE_WARNING)
```

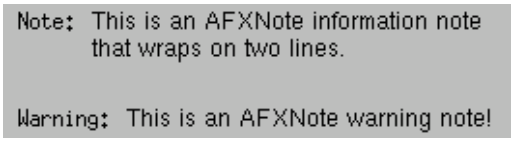


Figure 1: An example of a note and a warning from `AFXNote`.

Progress bar

The AFXProgressBar widget provides feedback during a process that takes a long time to complete.

For example,

```
pb = AFXProgressBar(parent, keyword, tgt,
    LAYOUT_FIX_HEIGHT|LAYOUT_FIX_WIDTH|
    FRAME_SUNKEN|FRAME_THICK|AFXPROGRESSBAR_SCANNER,
    0, 0, 200, 25)
```

If you want to control the display of the progress bar you can use the percentage or iterator mode and call setProgress with the appropriate value.

```
from abaqusGui import *
class MyDB(AFXDataDialog):
    ID_START = AFXDataDialog.ID_LAST
    def __init__(self, form):
        AFXDataDialog.__init__(self, form, 'My Dialog',
            self.OK|self.CANCEL, DECOR_RESIZE|DIALOG_ACTIONS_SEPARATOR)
        FXButton(self, 'Start Something', None, self, self.ID_START)
        FXMAPFUNC(self, SEL_COMMAND, self.ID_START, MyDB.onDoSomething)
        self.scannerDB = ScannerDB(self)
    def onDoSomething(self, sender, sel, ptr):
        self.scannerDB.create()
        self.scannerDB.showModal(self)
        getAFXApp().repaint()
        files = [
            'file_1.txt',
            'file_2.txt',
            'file_3.txt',
            'file_4.txt',
        ]
        self.scannerDB.setTotal( len(files) )
        for i in range( 1, len(files)+1 ):
            self.scannerDB.setProgress(i)
            # Do something with files[i]
        self.scannerDB.hide()
class ScannerDB(AFXDialog):
    def __init__(self, owner):
        AFXDialog.__init__(self, owner, 'Work in Progress',
            0, 0, DIALOG_ACTIONS_NONE)
        self.scanner = AFXProgressBar(self, None, 0,
            LAYOUT_FIX_WIDTH|LAYOUT_FIX_HEIGHT|
            FRAME_SUNKEN|FRAME_THICK|AFXPROGRESSBAR_ITERATOR,
            0, 0, 200, 22)
    def setTotal(self, total):
        self.scanner.setTotal(total)
    def setProgress(self, progress):
        self.scanner.setProgress(progress)
```



Note: The setProgress method has no effect on a progress bar that uses the scanner mode.

The progress bar has several different modes, as shown in *Figure 1*.

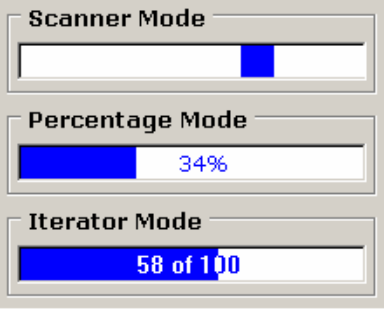


Figure 1: Three modes of the progress bar widget.

The `create` method

Most widgets in the Abaqus GUI Toolkit employ a two-stage creation process.

In the first stage the widget constructor builds the data structures for the widget. In the second stage the toolkit calls the widget's `create` method. The `create` method constructs all the windows required by the widget so that the widget can be displayed on the screen.

In most cases the application startup script first calls the constructors of all the widgets required to build the initial structure of an application by constructing the main window. The script then calls the application object's `create` method. This call traverses the entire widget hierarchy calling the `create` method of each widget. For more information and an example script, see *Startup script*.

If you create widgets after the startup script has called the application's `create` method, the `create` method must be called on those new widgets; otherwise, they will not be visible on the screen.

If your dialog box is posted by a form or by a procedure, the infrastructure calls the `create()` method on the dialog box. However, if you post a dialog box yourself, you must call the `create()` method on the dialog box before you call its `show()` method.

Similarly, if you construct icons that are used after a widget has been created, you must call the `create()` method on those icons before using them in a widget. For example, if you want to change a label's icon after it has already been shown in a dialog box, you must do the following:

1. Construct the new icon.
2. Call the new icon's `create()` method.
3. Pass the icon to the label using the label's `setIcon()` method.

Widgets and fonts

When the user starts an application, it sets the default font to be used for all widgets.
On Windows platforms the application obtains the default font from the desktop settings. On Linux platforms the default font is Helvetica.

The application can issue a command to change its default font. After the command is issued, all widgets created by the application use the new font. Alternatively, you can change an individual widget's font by using the `setFont` method that is available for many widgets.

You use the `getAFXFont` method to obtain the current font setting for a widget. Possible fonts are:

- FONT_PROPORTIONAL
- FONT_MONOSPACE
- FONT_BOLD
- FONT_ITALIC
- FONT_SMALL

The following example shows how you can change the default font for all widgets and the font for a particular widget:

```
# Get the current default font.
normalFont = getAFXApp().getNormalFont()

# Set the font to bold for subsequently created widgets.
getAFXApp().setNormalFont( getAFXFont(FONT_BOLD) )
FXLabel(self, 'Bold font')

# Restore the default font.
getAFXApp().setNormalFont(normalFont )

# Set the font of a widget after it is created.
l = FXLabel(self, 'Sample text')
l.setFont( getAFXFont(FONT_MONOSPACE) )
```

Layout managers

This section describes how to use the various layout managers in the Abaqus GUI Toolkit to arrange widgets in a dialog box.

In this section:

- *About layout managers*
- *Padding and spacing*
- *Horizontal and vertical frames*
- *Vertical alignment for composite children*
- *General-purpose layout managers*
- *Row and column layout manager*
- *Resizable regions*
- *Rotating regions*
- *Tab books*
- *Layout hints*
- *Layout examples*
- *Tips for specifying layout hints*

About layout managers

A layout manager places its children in a certain arrangement in its interior.
Layout managers use a combination of layout hints and packing styles to determine how to place and size their children. Layout managers in the Abaqus GUI Toolkit calculate relative sizes and relative positions, as opposed to absolute coordinates. This relative approach accounts automatically for changes such as different font sizes and window resizing.

The following layout managers are available in the Abaqus GUI Toolkit:

FXHorizontalFrame

Arranges widgets horizontally. For more information, see *Horizontal and vertical frames*.

FXVerticalFrame

Arranges widgets vertically. For more information, see *Horizontal and vertical frames*.

AFXVerticalAligner

Vertically aligns the first child of its children. For more information, see *Vertical alignment for composite children*.

FXPacker

Arranges widgets in a general manner. For more information, see *General-purpose layout managers*.

AFXDialog

Same capabilities as FXPacker. For more information, see *General-purpose layout managers*.

FXGroupBox

Same capabilities as FXPacker but allows a titled border. For more information, see *General-purpose layout managers*.

FXMatrix

Arranges widgets in rows and columns. For more information, see *Row and column layout manager*.

FXSplitter

Splits an area vertically or horizontally, and allows you to resize the areas. For more information, see *Resizable regions*.

FXSwitcher

Swaps children on top of each other (rotating regions). For more information, see *Rotating regions*.

FXTabBook

Displays one tab, or one page, of widgets at a time. The user selects the tab to view by clicking a tab button. For more information, see *Tab books*.

Padding and spacing

Layout managers (and most widgets) provide some default padding so that widgets are spaced apart from each other.

These values are commonly found near the end of the widget's list of arguments. For example,

```
FXPacker(…, pl, pr, pt, pb, …)
```

In general, you should accept the default values for padding. However, if you have nested layout managers, you should set the padding values to zero.

Layout managers also provide spacing between their children. These values are commonly found at the end of the widget's list of arguments. For example,

```
FXPacker(…, hs, vs)
```

In general, you should accept the default values for spacing.

Some “compound” widgets, such as `AFXTextField`, `AFXComboBox`, and `AFXSpinner`, have two padding values—one for the padding of the internal text field widget and a second for the entire widget that includes a label. You set the padding for the internal text field widget by passing padding values into the widget constructor. You set the padding for the entire widget by calling one of the padding methods on the widget; for example, `setPadLeft`.

Horizontal and vertical frames

The FXHorizontalFrame and FXVerticalFrame widgets arrange their children in rows or columns, respectively.

For example,

```
vf = FXVerticalFrame(parent)
FXButton(vf, 'Button 1')
FXButton(vf, 'Button 2')
FXButton(vf, 'Button 3')
```



Figure 1: An example of a vertical frame from FXVerticalFrame.

Vertical alignment for composite children

The AFXVerticalAligner widget is designed to align children that contain multiple children.

AFXVerticalAligner does the following:

1. Finds the maximum width of the first child of each of its children.
2. Sets the width of all the first children to the maximum width.

For example,

```
va = AFXVerticalAligner(parent)
AFXTextField(va, 16, 'Name:')
AFXTextField(va, 16, 'Address:')
AFXTextField(va, 16, 'Phone Number:')
```

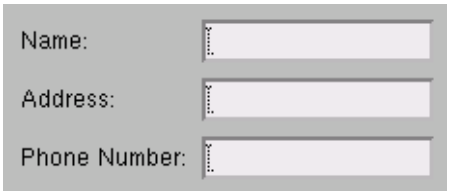


Figure 1: An example of vertical alignment from AFXVerticalAligner.

General-purpose layout managers

The Abaqus GUI Toolkit includes three general-purpose layout managers that have similar layout capabilities.

FXPacker

FXPacker is a general-purpose layout manager.

AFXDialog

AFXDialog provides similar capabilities to FXPacker. As a result, you do not need to provide a top-level layout manager as the first child in a dialog box; you can use the layout capabilities of the dialog box instead.

FXGroupBox

FXGroupBox provides the same capabilities as FXPacker. In addition, FXGroupBox can display a labeled border around its children. Abaqus/CAE uses the FRAME_GROOVE flag to produce a thin border around the children of the group box.

For example,

```
gb = FXGroupBox(parent, 'Render Style', FRAME_GROOVE)
FXRadioButton(gb, 'Wireframe')
FXRadioButton(gb, 'Filled')
FXRadioButton(gb, 'Shaded')
```

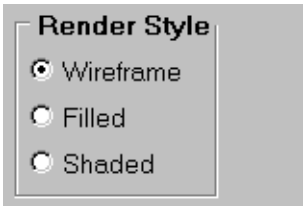


Figure 1: An example of a group box with a labeled border from FXGroupBox.

Row and column layout manager

The FXMatrix widget arranges its children in rows and columns.

You can perform the layout row-wise using the default value of the `opts` argument (MATRIX_BY_ROWS) or column-wise by setting `opts`=MATRIX_BY_COLUMNS. If you specify `opts`=MATRIX_BY_ROWS, the matrix will create the specified number of rows and as many columns as are needed to accommodate all its children. Conversely, if you specify `opts`=MATRIX_BY_COLUMNS, the matrix will create the specified number of columns and as many rows as are needed to accommodate all its children.

For example, using the default `opts`=MATRIX_BY_ROWS setting,

```
m = FXMatrix(parent, 2)
FXButton(m, 'Button 1')
FXButton(m, 'Button 2')
FXButton(m, 'Button 3')
FXButton(m, 'Button 4')
FXButton(m, 'Button 5')
FXButton(m, 'Button 6')
```



Figure 1: An example of a matrix with two rows from FXMatrix.

Resizable regions

The FXSplitter widget splits an area vertically or horizontally.

The user can drag the cursor on the region between the areas and resize the areas. For example,

```
sp = FXSplitter(parent,
    LAYOUT_FILL_X|LAYOUT_FIX_HEIGHT|SPLITTER_VERTICAL,
    0,0,0,100)
hf1 = FXHorizontalFrame(sp, FRAME_SUNKEN|FRAME_THICK)
FXLabel(hf1, 'This is area 1')
hf2 = FXHorizontalFrame(sp, FRAME_SUNKEN|FRAME_THICK)
FXLabel(hf2, 'This is area 2')
```

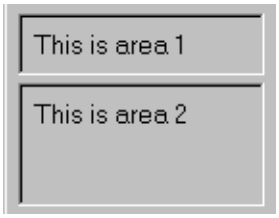


Figure 1: An example of resizable areas laid out vertically by FXSplitter.

Rotating regions

The FXSwitcher widget manages children that are positioned on top of each other.

FXSwitcher allows you to select which child should be shown by either sending it a message or calling its setCurrent method. When sending a message, you must set the message ID to FXSwitcher.ID_OPEN_FIRST for the first child. You must then increment the message ID from that value for the subsequent children, as shown in the following example. For more information on messages, see *Targets and messages*. To use the setCurrent method, you should provide the zero-based index of the child that you want to display. For example, to display the first child, you should call the `setCurrent` method with an index value of zero.

For example,

```
sw = FXSwitcher(parent)
FXRadioButton(hf, 'Option 1', sw, FXSwitcher.ID_OPEN_FIRST)
FXRadioButton(hf, 'Option 2', sw, FXSwitcher.ID_OPEN_FIRST+1)
hf1 = FXHorizontalFrame(sw)
FXButton(hf1, 'Button 1')
FXButton(hf1, 'Button 2')
hf2 = FXHorizontalFrame(sw)
FXButton(hf2, 'Button 3')
FXButton(hf2, 'Button 4')
```

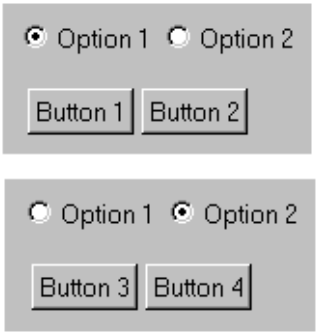


Figure 1: An example of a rotating region from FXSwitcher.

Tab books

The FXTabBook widget uses “tab items” to control the display of its “pages” one at a time.

FXTabBook expects that its odd-numbered children are FXTabItems and its even-numbered children are some type of layout manager. The layout manager contains whatever widgets are to be displayed in that page. Clicking a tab item will show the layout manager (and all its children) associated with that tab while hiding all the other layout managers. Typically, a horizontal or vertical frame is used for the layout manager, and its frame options are set to FRAME_RAISED | FRAME_THICK to provide a standard border.

You can nest tab books to provide tabs within tabs, as shown in the following example:

```
tabBook1 = FXTabBook(self, None, 0, LAYOUT_FILL_X)
FXTabItem(tabBook1, 'Tab Item 1')
tab1Frame = FXHorizontalFrame(tabBook1,
    FRAME_RAISED|FRAME_SUNKEN)
FXLabel(tab1Frame, '
    This is the region controlled by Tab Item 1.')
FXTabItem(tabBook1, 'Tab Item 2')
tab2Frame = FXHorizontalFrame(tabBook1, FRAME_RAISED|FRAME_SUNKEN)

tabBook2 = FXTabBook(tab2Frame, None, 0,
    TABBOOK_LEFTTABS|LAYOUT_FILL_X)
FXTabItem(tabBook2, 'Subtab Item 1', None, TAB_LEFT)
subTab1Frame = FXHorizontalFrame(tabBook2,
    FRAME_RAISED|FRAME_SUNKEN)
AFXNote(subTab1Frame,
    'This is a note\nin sub-tab item 1\nthat extends\n' \
    'over several\nlines.')
FXTabItem(tabBook2, 'Subtab Item 2', None, TAB_LEFT)
subTab2Frame = FXHorizontalFrame(tabBook2,
    FRAME_RAISED|FRAME_SUNKEN)
```

Figure 1 shows an example of nested tab books.

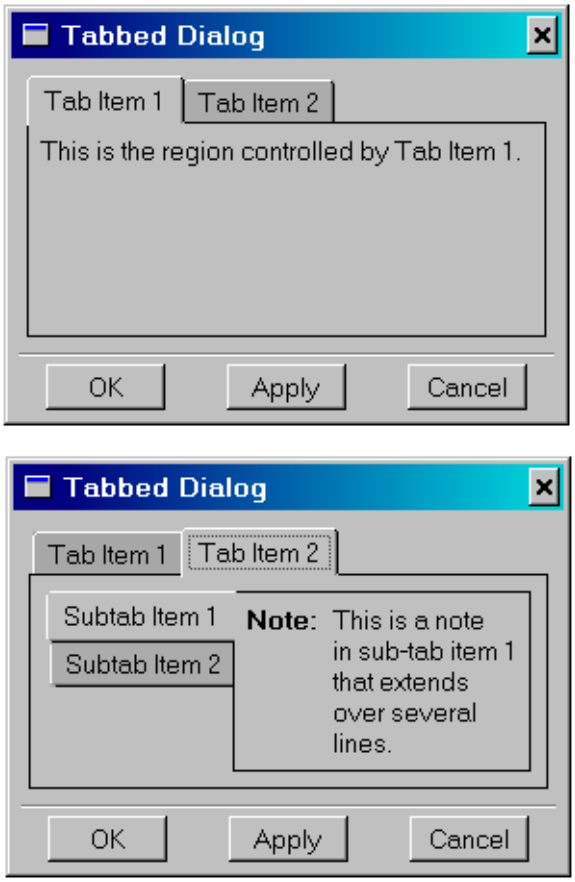


Figure 1: An example of two subtab pages.

Layout hints

The FXPacker, FXTopWindow, and FXGroupBox widgets accept the layout hints in their children.

The following layout hints are accepted:

LAYOUT_SIDE_TOP

Attaches a widget to the top side of the cavity. LAYOUT_SIDE_TOP is the default layout hint.

LAYOUT_SIDE_BOTTOM

Attaches a widget to the bottom side of the cavity.

LAYOUT_SIDE_LEFT

Attaches a widget to the left side of the cavity.

LAYOUT_SIDE_RIGHT

Attaches a widget to the right side of the cavity.

You should specify only one of the LAYOUT_SIDE_* hints per child. The top and bottom hints effectively reduce the height of the available space remaining to place other children. The left and right hints effectively reduce the width of the available space remaining to place other children.

All layout managers support the following layout hints:

- LAYOUT_LEFT (default) and LAYOUT_RIGHT. The layout manager places the widget on the left or right side of the space remaining in the container.
- LAYOUT_TOP (default) and LAYOUT_BOTTOM. The layout manager places the widget on the top or bottom side of the space remaining in the container.
- LAYOUT_CENTER_X and LAYOUT_CENTER_Y. The layout manager centers the widget in the *X*- or *Y*-direction in the parent. The manager adds extra spacing around the widget to place it at the center of the space available to it. The widget's size will be its default size unless you specify LAYOUT_FIX_WIDTH or LAYOUT_FIX_HEIGHT.
- LAYOUT_FILL_X and LAYOUT_FILL_Y. You can specify either none, one, or both of these layout hints. LAYOUT_FILL_X causes the parent layout manager to stretch or to shrink the widget to accomodate the available space. If you place more than one child with this option side by side, the manager subdivides the available space proportionally to the children's default size. LAYOUT_FILL_Y has the identical effect in the vertical direction.

FXPacker, FXTopWindow, and FXGroupBox must use LAYOUT_LEFT and LAYOUT_RIGHT with LAYOUT_SIDE_TOP and LAYOUT_SIDE_BOTTOM. The Abaqus GUI Toolkit ignores hints if they do not make sense; for example, FXHorizontalFrame ignores LAYOUT_TOP and LAYOUT_BOTTOM. Similar rules apply for the other hints.

The majority of widgets in the Abaqus GUI Toolkit have width and height arguments in their constructors. In most cases you can accept the default value of zero for these arguments, which allows the application to determine the proper size of the widget. However, in some cases you will need to set specific values for the width and height of a widget. To set the width and height, you must pass the LAYOUT_FIX_WIDTH and LAYOUT_FIX_HEIGHT flags to the options argument of the widget. If you do not pass these flags to the options argument, the toolkit will ignore the values that you specified for the width and height.

Layout hints are described in detail in *Layout hints*.

Layout examples

The following examples create three buttons, one at a time, using the default layout hints.

As each button is created, the figures show the effect on the space remaining in the layout cavity.

Example 1

The first example starts by creating a single button on the left side of the cavity. The default value for the vertical position is LAYOUT_TOP, so the example places the button on the left side and at the top of the available space.

```
gb = FXGroupBox(parent, '')
FXButton(gb, 'Button 1', opts=LAYOUT_SIDE_LEFT|BUTTON_NORMAL)
```

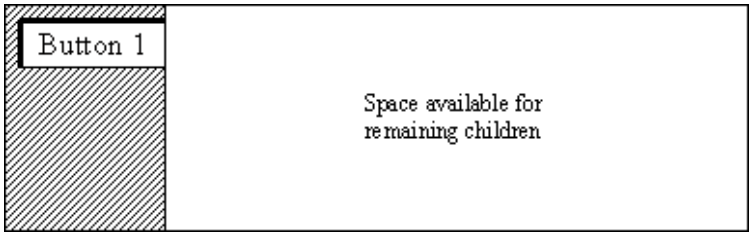


Figure 1: Creating a button on the left side and at the top of the layout cavity.

The following statement adds a second button on the left side at the top of the available space:

```
FXButton(gb, 'Button 2', opts=LAYOUT_SIDE_LEFT|BUTTON_NORMAL)
```

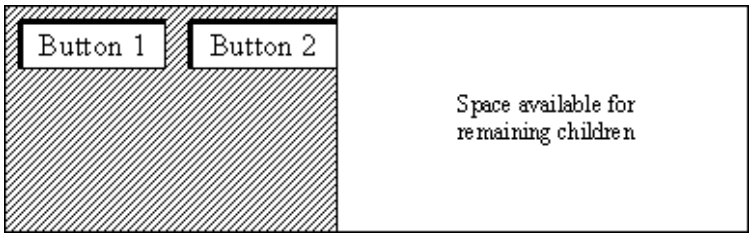


Figure 2: Adding a second button on the left side at the top of the layout cavity.

The following statement adds a third button on the left side at the top of the available space:

```
FXButton(gb, 'Button 3',
    opts=LAYOUT_SIDE_LEFT|BUTTON_NORMAL)
```

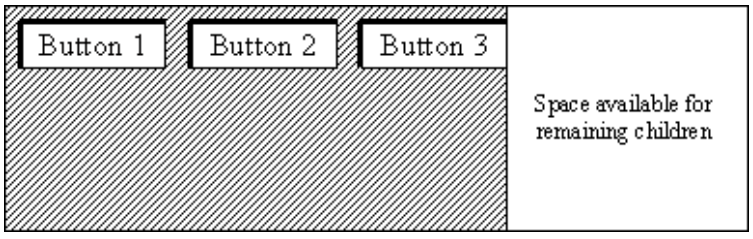


Figure 3: Adding a third button on the left side at the top of the layout cavity.

Figure 4 shows the final configuration of the three buttons.



Figure 4: The final configuration of the buttons.

Example 2

The second example illustrates how you can use nondefault layout hints. The example starts by using the default hints to position a button on top of the available space and on the left.

```
gb = FXGroupBox(p,'')
FXButton(gb,'Button 1')
```



Figure 5: Creating a button on the left side and at the top of the layout cavity.

The example then positions a second button on the right side on the bottom of the layout cavity.

```
FXButton(gb, 'Button 2',
    opts=LAYOUT_SIDE_BOTTOM|LAYOUT_RIGHT|BUTTON_NORMAL)
```



Figure 6: Adding a second button on the right side at the bottom of the layout cavity.

Finally, the example places a third button on the bottom of the available space and centered in the *X*-direction.

```
FXButton(gb, 'Button 3',
    opts=LAYOUT_SIDE_BOTTOM|LAYOUT_CENTER_X|BUTTON_NORMAL)
```



Figure 7: Adding a third button in the center at the bottom of the layout cavity.

Figure 8 shows the final configuration of the three buttons.

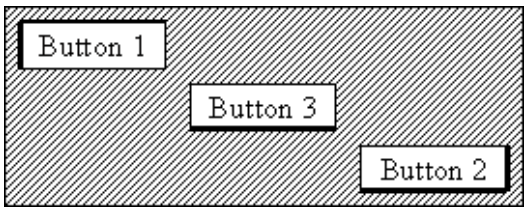


Figure 8: The final configuration of the three buttons.

Tips for specifying layout hints

This section provides tips for specifying layout hints supported by the layout managers in the Abaqus GUI Toolkit.

- Do not over specify layout hints. In many cases the default values are what you want, and you do not need to specify the hints.
- Think in terms of simple rows and columns, and use horizontal or vertical frames whenever possible.
- To avoid building up excessive padding, set the padding to zero in nested layout managers.
- Layout hints are described in detail in *Layout hints*.

Dialog boxes

This section describes the dialog boxes that you can create using the Abaqus GUI Toolkit.

In this section:

- *About dialog boxes*
- *Modal versus modeless*
- *Showing and hiding dialog boxes*
- *Message dialog boxes*
- *Custom dialog boxes*
- *Data dialog boxes*
- *Common dialog boxes*

About dialog boxes

The following general types of dialog boxes are available in the Abaqus GUI Toolkit: message, custom, data, and common.

Message dialog boxes

Message dialog boxes allow you to post error, warning, or informational messages.

Custom dialog boxes

Custom dialog boxes allow you to build any custom interface. However, you must supply the infrastructure needed to make the dialog box behave as required.

Data dialog boxes

Data dialog boxes provide support for dialog boxes in which users enter data. Data dialog boxes are designed to supply user inputs to forms, which automatically issue commands. For more information, see *Form modes*.

Common dialog boxes

Common dialog boxes are dialog boxes that provide standard functionality commonly found in many applications. The **File Selection** dialog box is a typical common dialog box.

These dialog boxes, along with other details related to dialog box construction and behavior, are described in this chapter.

Modal versus modeless

A dialog box can be either modal or modeless.

Modal

A modal dialog box prevents interaction with the rest of the application until the user dismisses the dialog box.

Modeless

A modeless dialog box allows the user to interact with other parts of the GUI while the dialog box is posted. In Abaqus/CAE all secondary dialog boxes except for tips should be modal dialog boxes.

A dialog box itself is not defined as modal or modeless—the behavior is obtained from the method used to post the dialog box.

For dialog boxes posted by forms, you can set the modal behavior by calling the form's `setModal` method and providing an argument of either True or False. If you call `setModal` with True as its argument, the form will post the next dialog box modally. You can call the `setModal` method several times within one form if you need to change the modal behavior between the various dialog boxes managed by the form.

For dialog boxes that you post yourself, you can use the `showModal` method instead of the `show` method described in the next section. *File/Directory selector* includes an example that uses the `showModal` method.

Showing and hiding dialog boxes

Dialog boxes have `show` and `hide` methods that post or unpost the dialog box from the screen.

In most cases you do not need to call these methods because the mode infrastructure calls them for you. However, you may want to write your own `show` and `hide` methods to perform some special processing that will be executed just before your application posts or unposts the dialog box. For example, you can register and unregister queries inside the `show` and `hide` methods. You must call the base class versions of the `show` and `hide` methods, or the methods will not behave as expected. For example, in your dialog class code you could add the following lines:

```
def show(self):

    # Do some special processing here.
    ...

    # Call base class method.
    AFXDataDialog.show(self)

def hide(self):

    # Do some special processing here.
    ...

    # Call base class method.
    AFXDataDialog.hide(self)
```

Message dialog boxes

The AFXMessageDialog class extends the FXMessageDialog class by enforcing certain characteristics of the dialog box; for example, the window title and message symbol.
These characteristics make message dialog boxes in Abaqus/CAE consistent and easy to use. This section describes the message dialog boxes that you can create with the Abaqus GUI Toolkit.

In this section:

- *Error dialog boxes*
- *Warning dialog boxes*
- *Information dialog boxes*
- *Specialized message dialog boxes*

Error dialog boxes

You post error dialog boxes in response to a failure condition that the application cannot resolve.

Error dialog boxes have the following characteristics:

- The application name is displayed in their title bar.
- An error symbol is displayed on the left side of the dialog box.
- The action area contains only a **Dismiss** button.
- They are modal.

For example:

```
mainWindow = getAFXApp().getAFXMainWindow()
showAFXErrorDialog(mainWindow, 'An invalid value was supplied.')
```

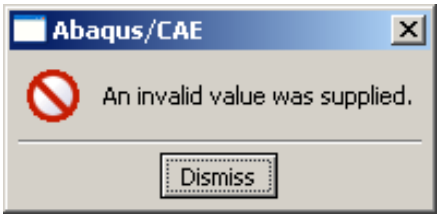


Figure 1: An example of an error dialog box from `showAFXErrorDialog`.

Warning dialog boxes

You post warning dialog boxes in response to a condition that the application needs user assistance to resolve.

Warning dialog boxes have the following characteristics:

- The application name is displayed in their title bar.
- A warning symbol is displayed on the left side of the dialog box.
- The action area may contain **Yes**, **No**, and **Cancel** buttons.
- They are modal.

To find out which button in the warning dialog box was pressed by the user, you must pass the warning dialog box a target and a selector and you must create a message map entry in the form to handle that message. In your message handler you can query the warning dialog box using the `getPressedButtonId` method. The following examples illustrate how to create a warning dialog box:

You must define an ID in the form class:

```
from abaqusGui import *
class MyForm(AFXForm):
    [
        ID_WARNING,
    ] = range(AFXForm.ID_LAST, AFXForm.ID_LAST+1)

    def __init__(self, owner):

        # Construct the base class.
        #
        AFXForm.__init__(self, owner)

        FXMAPFUNC(self, SEL_COMMAND, self.ID_WARNING,
            MyForm.onCmdWarning)

        ...

    def doCustomChecks(self):

        if <someCondition>:
            showAFXWarningDialog( self.getCurrentDialog(),
                'Save changes made in the dialog?',
                AFXDialog.YES | AFXDialog.NO,
                self, self.ID_WARNING)
            return False

        return True

    def onCmdWarning(self, sender, sel, ptr):

        if sender.getPressedButtonId() == \
            AFXDialog.ID_CLICKED_YES:
                self.issueCommands()
        elif sender.getPressedButtonId() == \
            AFXDialog.ID_CLICKED_NO:
                self.deactivate()
```

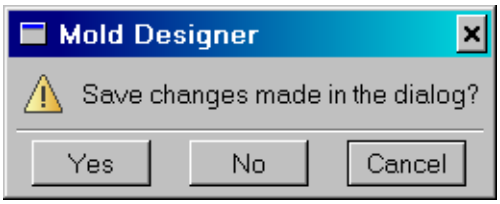


Figure 1: An example of a warning dialog box from showAFXWarningDialog.

There are two other variations of warning dialog boxes:

- showAFXDismissableWarningDialog
- showAFXItemsWarningDialog

The dialog box created by showAFXDismissableWarningDialog contains a check button that allows the user to specify whether the application should continue to post the warning dialog box each time the warning occurs. You can check the state of the button by calling the getCheckButtonState method of the warning dialog.

The dialog box created by showAFXItemsWarningDialog contains a scrolled list of items to be displayed to the user. The list prevents the dialog box from becoming too tall when it is displaying a long list of items.

Information dialog boxes

You post information dialog boxes to provide an explanatory message.

Information dialog boxes have the following characteristics:

- The application name is displayed in their title bar.
- An information symbol is displayed on the left side of the dialog box.
- The action area contains only a **Dismiss** button.
- They are modal.

For example,

```
mainWindow = getAFXApp().getAFXMainWindow()
showAFXInformationDialog(mainWindow,
    'This is an information dialog.')
```



Figure 1: An example of an information dialog box from `showAFXInformationDialog`.

Specialized message dialog boxes

If you need more flexibility than the standard message dialog boxes, you must derive a new dialog box from AFXDialog and provide the specialized handling.

For more information, see *Custom dialog boxes*.

Custom dialog boxes

AFXDialog is the base class for the other dialog box classes in the toolkit.
If none of the other dialog box classes suit your needs, you must derive your dialog box from AFXDialog and provide most of the dialog processing yourself. This section describes how you can use AFXDialog to create custom dialog boxes.

In this section:

- *About custom dialog boxes*
- *Constructors*
- *Sizing and location*
- *Action area*
- *Custom action area button names*
- *Action button handling*

About custom dialog boxes

AFXDialog is the base class for the other dialog box classes in the toolkit.
If none of the other dialog box classes suit your needs, you must derive your dialog box from AFXDialog and provide most of the dialog box processing yourself.

The AFXDialog class extends the FXDialog class by providing the following features:

- Button flags that allow the automatic construction of action area buttons.
- Option flags that control the placement of the action area. Option flags also determine whether to include a separator between the action area and the rest of the dialog box.
- Message IDs for the various action area commit semantics.
- Methods to add action area buttons manually.
- Automatic handling of the **No**, **Cancel**, and **Dismiss** buttons. Automatic handling is also provided for the **Close** (X) button on the right hand side of the dialog box's title bar.
- Automatic destruction of the dialog box after it is unposted.

See *Action area* for more details.

Constructors

There are three prototypes of the AFXDialog constructor.

The difference between the three prototypes is the occluding behavior of the dialog box, as illustrated in the following examples:

- The following statement creates a dialog box that always occludes the main window when overlapping with the main window:

```
AFXDialog(title, actionButtonIds=0,
    opts=DIALOG_NORMAL, x = 0, y = 0, w = 0, h = 0)
```

- The following statement creates a dialog box that always occludes its owner widget (usually a dialog box) when overlapping with the widget:

```
AFXDialog(owner, title, actionButtonIds=0,
    opts=DIALOG_NORMAL, x = 0, y = 0, w = 0, h = 0)
```

- The following statement creates a dialog box that can be occluded by any other windows in the application:

```
AFXDialog(app, title, actionButtonIds=0,
    opts = DIALOG_NORMAL, x = 0, y = 0, w = 0, h = 0)
```

When you construct a dialog box, you will start by deriving from the AFXDialog class. The first thing you should do in the constructor body is call the base class constructor to properly initialize the dialog. Then, you would build the contents of your dialog by adding widgets. For example:

```
class MyDB(AFXDialog):

    # My constructor
    def __init__(self):

        # Call base class constructor
        AFXDialog.__init__(self, 'My Dialog', self.DISMISS)

    # Add widgets next...
```

Sizing and location

By default, the user cannot resize a dialog box. However, if a dialog box contains text fields or lists that can be stretched to show more entries, the user should be allowed to resize the dialog box.

Resizing can be allowed by specifying the DECOR_RESIZE flag in the dialog box constructor.



Note: Dialog boxes created by AFXDialog do not support minimizing and maximizing; they ignore these flags if they are included in the dialog box constructor.

You should never specify the size and location of the dialog box in its constructor. The Abaqus GUI Toolkit will place the dialog box on the screen and determine its proper size.

Action area

The action area of a dialog box contains buttons, such as **OK** and **Cancel**. These buttons allow the user to commit values from the dialog box, to close the dialog box, or to perform some other action.

AFXDialog supports the automatic creation of an action area and its buttons through the use of bit flags in the dialog box constructor. You can use the flags described in *Table 1* to include standard action area buttons.

Table 1: Action area flags.

Button flag	Message ID	Label	Semantics
AFXDialog.OK	AFXDialog.ID_CLICKED_OK	OK	Commit the values in the dialog box, process them, and then hide the dialog box.
AFXDialog.CONTINUE	AFXDialog.ID_CLICKED_CONTINUE	Continue…	Commit the values in the dialog box, hide it, and continue collecting input from the user in another dialog box or prompt.
AFXDialog.APPLY	AFXDialog.ID_CLICKED_APPLY	Apply	Same as OK, except the dialog box is not hidden.
AFXDialog.DEFAULTS	AFXDialog.ID_CLICKED_DEFAULTS	Defaults	Reset the values in the dialog box to their defaults.
AFXDialog.YES	AFXDialog.ID_CLICKED_YES	Yes	Invoke the affirmative action in response to the question posed by the dialog box.
AFXDialog.NO	AFXDialog.ID_CLICKED_NO	No	Invoke the negative action in response to the question posed by the dialog box.
AFXDialog.CANCEL	AFXDialog.ID_CLICKED_CANCEL	Cancel	Do not commit the values in the dialog box; just hide the dialog box. Optionally, for the AFXDataDialog a bailout may be posted if the user has changed any values since the last commit.
AFXDialog.DISMISS	AFXDialog.ID_CLICKED_DISMISS	Dismiss	Hide the dialog box without taking any other action.

AFXDialog also supports the following options that determine the location of the action area:

DIALOG_ACTIONS_BOTTOM

This option places the action area at the bottom of the dialog box and is the default option.

DIALOG_ACTIONS_RIGHT

This option places the action area on the right side of the dialog box.

DIALOG_ACTIONS_NONE

This option does not create an action area; for example, in a toolbox dialog box.

You can also specify whether a separator should be placed between the action area and the rest of the dialog box by including the following flag in the options:

DIALOG_ACTIONS_SEPARATOR

The style in Abaqus/CAE is to omit a separator if there is already delineation between the action area and the rest of the dialog box; for example, a frame that stretches across the entire width of the dialog box along the bottom of the dialog box. The following statements illustrate how you define an action area in a dialog box with a separator:

```
class ActionAreaDB(AFXDialog):
```

```
    def __init__(self):

        AFXDialog.__init__(self, 'Action Area Example1',
            self.OK|self.APPLY|self.CANCEL,
            DIALOG_ACTIONS_SEPARATOR)

        FXLabel(self, 'Standard action area example dialog.')
```

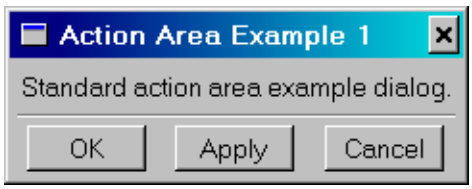


Figure 1: An example of a standard action area.

Custom action area button names

You can use the se the `appendActionButton` method to add action area buttons.

The flags in *Table 1* cover all the semantics you might need in a dialog box. As a result, there is no need for any additional custom flags; however, there may be cases where you want to use a different label for one of the standard actions. To use a different label for one of the standard actions, you do not specify any button flags in the constructor arguments; you use the `appendActionButton` method to add your own action area buttons. The `appendActionButton` method has two prototypes:

```
appendActionButton(buttonId)
appendActionButton(text, tgt, sel)
```

The first version of the prototype creates a standard action area button as defined in *Table 1*. The second version of the prototype creates a button whose label is given as the text argument. In addition, the second version allows you to set the target and selector so that you can catch messages from this button and act accordingly. The following statements show how you can create custom action area buttons:

```
class ActionAreaDB(AFXDialog):
    def __init__(self):

        AFXDialog.__init__(self, 'Action Area Example 2',
            0, DIALOG_ACTIONS_SEPARATOR)
        FXLabel(self, 'Custom action area example dialog.')
        self.appendActionButton('Highlight', self,
            self.ID_CLICKED_APPLY)
        self.appendActionButton(self.CANCEL)
```

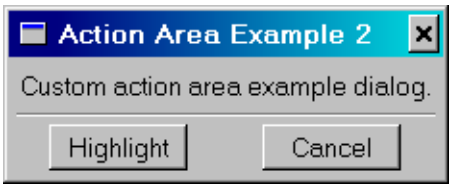


Figure 1: An example of a custom action area.

Action button handling

AFXDialog and AFXDataDialog provide some automatic handling of the messages that are sent when a button in the action area is clicked. If you want to perform some actions other than those provided by the dialog box, you must catch the messages sent by the action area buttons and write your own message handler.

For example, if you want to take an action when the user clicks the **Apply** button in the dialog box, you must catch the (ID_CLICKED_APPLY | SEL_COMMAND) message and map it to a message handler in your dialog box. For more information, see *Targets and messages*.

Data dialog boxes

A data dialog box is a dialog box in which data are collected from the user. In contrast, a message dialog box displays only a message and a toolbox just holds buttons. This section describes how you can create a data dialog box.

In this section:

- *About data dialog boxes*
- *Constructors*
- *Bailout*
- *Constructor contents*
- *Transitions*
- *Updating your GUI*
- *Action area*

About data dialog boxes

A data dialog box is a dialog box in which data are collected from the user.
In contrast, a message dialog box displays only a message, and a toolbox just holds buttons. AFXDataDialog is designed to be used in conjunction with a mode to gather data from the user. The data are then processed in a command. You should use AFXDataDialog if you need to issue a command. You should also use AFXDataDialog if the dialog box belongs to a module or nonpersistent toolset so that the GUI infrastructure can properly manage the dialog box when the user switches modules.

The AFXDataDialog class is derived from AFXDialog and provides the following additional features:

- A bailout mechanism.
- Standard action area button behavior designed to work with a form.
- Keyword usage.
- Transitions that define GUI state changes in the dialog box.

Constructors

There are two prototypes of the AFXDataDialog constructor.

The difference between the two prototypes is the occluding behavior of the dialog box, as illustrated in the following examples:

- The following statement creates a dialog box that always occludes the main window when overlapping with the main window:

```
AFXDataDialog(mode, title, actionButtonIds=0,
    opts=DIALOG_NORMAL, x = 0, y = 0, w = 0, h = 0 )
```

- The following statement creates a dialog box that always occludes its owner widget (usually a dialog box) when overlapping with the widget.

```
AFXDataDialog(mode, owner, title, actionButtonIds=0,
    opts=DIALOG_NORMAL, x = 0, y = 0, w = 0, h = 0 )
```

When you construct a dialog box, you will start by deriving from the AFXDataDialog class. The first thing you should do in the constructor body is call the base class constructor to properly initialize the dialog. Then, you would build the contents of your dialog by adding widgets. For example:

```
class MyDB(AFXDataDialog):

    # My constructor
    def __init__(self):

        # Call base class constructor
        AFXDataDialog.__init__(self, form,  'My Dialog',
            self.OK|self.CANCEL)

        # Add widgets next...
```

When a dialog box is unposted, it is removed from the screen. By default, a dialog box is deleted when it is unposted. Deleting a dialog box removes both the GUI resources associated with the dialog box and the dialog box's data structures. In contrast, you can choose to destroy a dialog box when it is unposted. Destroying a dialog box removes only the GUI resources and retains the dialog box's data structures.

If there is some dialog box GUI state that you want to retain between postings of the dialog box, you should specify that the dialog box is destroyed only when it is unposted. Therefore, when the dialog box is posted again, it retains its data structures and the old state is still intact. For example, assume that your dialog box contains a table and the user resizes one of the columns of the table. If you only destroy the dialog box when it is unposted, the table column sizes will be remembered the next time the dialog box is posted. To specify that a dialog box should be destroyed when unposted, add the DIALOG_UNPOST_DESTROY flag to the dialog box constructor's `opts` argument.

Bailout

AFXDataDialog supports automatic bailout handling through the specification of a bit flag in the dialog box constructor.

If you request bailout processing and the user changes some values in the dialog box and presses **Cancel**, the application posts a standard warning dialog box. The following statement requests bailout processing:

```
AFXDataDialog.__init__(self, form, 'Create Part',
    self.OK|self.CANCEL,
    DIALOG_ACTIONS_SEPARATOR|DATADIALOG_BAILOUT)
```

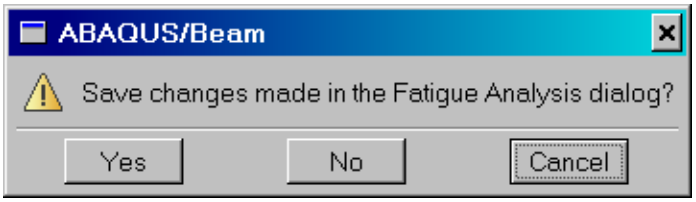


Figure 1: An example of a bailout.

After the standard warning dialog box has been posted, the behavior is as follows:

- If the user clicks **Yes** from the standard warning dialog box, the data dialog box will be processed as if the user had originally pressed **OK**.
- If the user clicks **No** from the standard warning dialog box, the data dialog box will be unposted without any processing.
- If the user clicks **Cancel** from the standard warning dialog box, the data dialog box will remain posted and no action will be taken.

Constructor contents

You use the constructor of the dialog box to create the widgets that will appear in the dialog box.

To keep the GUI up-to-date with the application state and vice versa, you use keywords as targets of widgets. Keywords are defined as members of a form, and the form is passed to the dialog box as a dialog box constructor argument. For more information, see *AFXKeywords*. The following script shows how you can use keywords to construct a dialog box. *Figure 1* shows the **Graphics Options** dialog box generated by the example script.

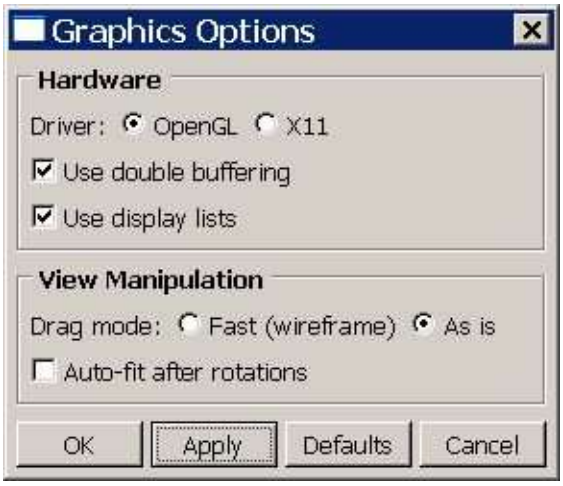


Figure 1: Graphics Options data dialog box.

```
class GraphicsOptionsDB(AFXDataDialog):

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self, form):

        AFXDataDialog.__init__(self, form, 'Graphics Options',
            self.OK|self.APPLY|self.DEFAULTS|self.CANCEL)

        # Hardware frame
        #
        gb = FXGroupBox(self, 'Hardware',
            FRAME_GROOVE|LAYOUT_FILL_X)
        hardwareFrame = FXHorizontalFrame(gb,
            0, 0,0,0,0, 0,0,0,0)
        FXLabel(hardwareFrame, 'Driver:')
        FXRadioButton(hardwareFrame, 'OpenGL',
            form.graphicsDriverKw, OPEN_GL.getId())
        FXRadioButton(hardwareFrame, 'X11',
            form.graphicsDriverKw, X11.getId())
        FXCheckButton(gb, 'Use double buffering',
            form.doubleBufferingKw)
        displayListBtn = FXCheckButton(gb, 'Use display lists',
            form.displayListsKw)

        # View Manipulation frame
        #
        gb = FXGroupBox(self, 'View Manipulation',
            FRAME_GROOVE|LAYOUT_FILL_X)
        hf = FXHorizontalFrame(gb, 0, 0,0,0,0, 0,0,0,0)
        FXLabel(hf, 'Drag mode:')
        FXRadioButton(hf, 'Fast (wireframe)', form.dragModeKw,
```

```
            FAST.getId())
        FXRadioButton(hf, 'As is', form.dragModeKw,
            AS_IS.getId())
        FXCheckButton(gb, 'Auto-fit after rotations',
            form.autoFitKw)
```

Transitions

Transitions provide a convenient way to change the GUI state in a dialog box. Transitions are used to stipple widgets or to rotate regions when some other control in the dialog box is activated.
If the behavior in your dialog box can be described in terms of simple transitions, you can use the `addTransition` method to produce the state changes.

Transitions compare the value of a keyword with a specified value. If the operator condition is met, a message is sent to the specified target object. Transitions have the following prototype:

```
addTransition(keyword,
operator, value, tgt, sel, ptr)
```

For example, when the user selects **Wireframe** as the render style in the **Part Display Options** dialog box, Abaqus/CAE does the following:

- Stipples the **Show dotted lines in hidden render style** button.
- Stipples the **Show edges in shaded render style** button.
- Checks the **Show silhouette edges** button.

These transitions can be described as follows:

- If the value of the render style keyword equals WIREFRAME, send the **Show dotted lines...** button an ID_DISABLE message.
- If the value of the render style keyword equals WIREFRAME, send the **Show edges in shaded...** button an ID_DISABLE message.
- If the value of the render style keyword equals WIREFRAME, send the **Show silhouette edges** button an ID_ENABLE message.

You can write these transitions with the Abaqus GUI Toolkit as follows:

```
self.addTransition(form.renderStyleKw, AFXTransition.EQ,
    WIREFRAME.getId(), showDottedBtn,
    MKUINT(FXWindow.ID_DISABLE, SEL_COMMAND), None)

self.addTransition(form.renderStyleKw, AFXTransition.EQ,
    WIREFRAME.getId(), showEdgesBtn,
    MKUINT(FXWindow.ID_DISABLE, SEL_COMMAND), None)

self.addTransition(form.renderStyleKw, AFXTransition.EQ,
    WIREFRAME.getId(), showSilhouetteBtn,
    MKUINT(FXWindow.ID_ENABLE, SEL_COMMAND), None)
```

You can also pass additional user data to the object using the last argument of the `addTransition` method. *Figure 1* shows an example that uses transitions to control how the application stipples widgets.

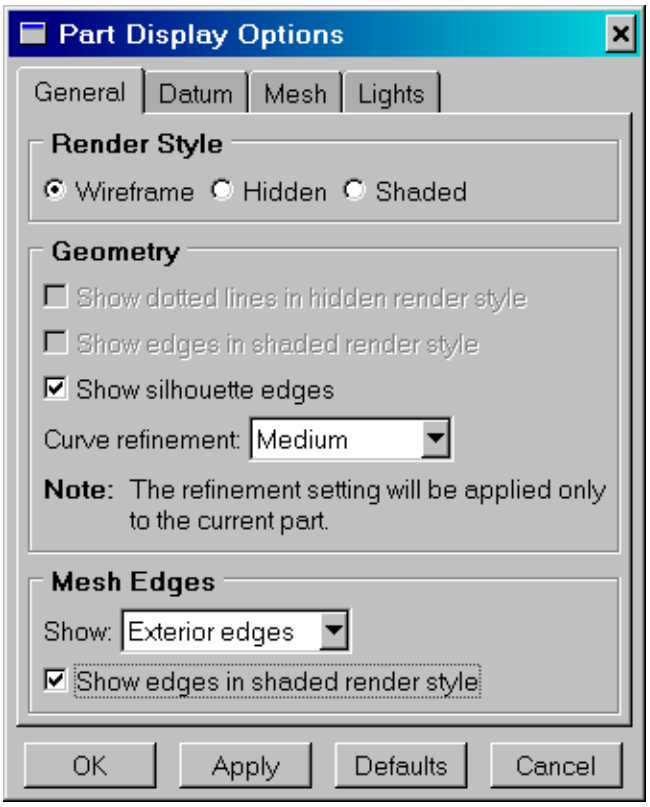


Figure 1: An example of using transitions to control how the application stipples widgets.

Updating your GUI

If the GUI behavior of your dialog box cannot be described in terms of simple transitions (for example, if you need to stipple a button based on the setting of two other buttons), you can use the `processUpdates` method to update your GUI.

The `processUpdates` method is called during each GUI update cycle, so you should not do anything that is time consuming in this method. Generally, you should perform tasks such as enabling and disabling, or showing and hiding widgets. For example:

```
def processUpdates(self):

    if self.form.kw1.getValue() == 1 and \
       self.form.kw2.getValue() == 2:

            self.btn1.disable()
    else:
            self.btn1.enable()
```

If the tasks you need to perform are time consuming, you should write your own message handler that is invoked only upon some specific user action. For example, if you need to scan an ODB for valid data, you could make the commit button of the dialog send a message to your dialog box. That message would invoke your message handler that does the scanning. That way, the scanning occurs only when the user commits the dialog, not during every GUI update cycle. For more information on message handlers, see *Targets and messages*.

Action area

The AFXDataDialog class provides standard handling for all the buttons that can appear in the action area.

Table 1 shows the action that the application takes when each of these buttons is clicked.

Table 1: Action area buttons.

Button	Action
OK	Send the form an (ID_COMMIT, SEL_COMMAND) message and its button ID.
Apply	Send the form an (ID_COMMIT, SEL_COMMAND) message and its button ID.
Continue	Send the form an (ID_GET_NEXT, SEL_COMMAND) message.
Defaults	Send the form an (ID_SET_DEFAULTS, SEL_COMMAND) message.
Cancel	Check for bailout, send the form an (ID_DEACTIVATE, SEL_COMMAND) message.
"**x**" in title bar	Perform the **Cancel** button action.

If your dialog has more than one "apply" button, you can handle this by routing messages from the button to the apply message handler in the form. In the form, you can use the `getPressedButtonId` method to determine which button was pressed and take the appropriate action. For example, in your dialog constructor:

```
self.appendActionButton('Plot', self, self.ID_PLOT)
FXMAPFUNC(self, SEL_COMMAND, self.ID_PLOT,
    AFXDataDialog.onCmdApply)
self.appendActionButton('Highlight', self, self.ID_HIGHLIGHT)
FXMAPFUNC(self, SEL_COMMAND, self.ID_HIGHLIGHT,
    AFXDataDialog.onCmdApply)
```

and in your form code:

```
def doCustomChecks(self):

    if self.getPressedButtonId() == self.getCurrentDialog().ID_PLOT:
        # Enable plot commands, disable highlight commands
    else:
        # Enable highlight commands, disable plot commands
    return True
```

Common dialog boxes

The Abaqus GUI Toolkit provides some pre-built dialog boxes for handling common operations. This section provides details on how to use these dialog boxes.

In this section:

- *File/Directory selector*
- *Print dialog box*
- *Color selector dialog box*

File/Directory selector

The **File Selector** dialog box is used to gather a file or directory name from the user.

It has the following characteristics:

- The title bar can be set.
- The file filters can be set.
- The following error checking is provided:
  - Check to see if the file exists.
  - Check for proper permissions.
  - Check to see if the selection is a file.
- Allows read-only access.
- Accepts keywords and a target.

The file selection dialog box has the following prototypes:

```
AFXFileSelectorDialog(form, title, fileNameKw,
    readOnlyKw, opts, patterns, patternIndexTgt)

AFXFileSelectorDialog(parent, title, fileNameKw,
    readOnlyKw, opts, patterns, patternIndexTgt)
```

You use the first constructor when you have a form associated with the dialog box that issues a command; for example, the dialog box that appears when you click **File->Open Database**. You use the second constructor when the dialog box collects input from the user to be used in another dialog box. For example, when printing to a file from the **Print** dialog box, the user is presented with a text field to enter a file name and a **Select** button. The **Select** button posts a file selection dialog box that returns the selected file to the **Print** dialog box but does not issue any command.

You must create the `fileNameKw` argument using the AFXStringKeyword method. Similarly, you must create the `readOnlyKw` argument using the AFXBoolKeyword method. If the user clicks **OK**, the file selection dialog box automatically updates the `fileNameKw` and `readOnlyKw` arguments. In addition, when the dialog box is posted, it will set the current directory based on the path of the `fileNameKw` argument. This means that the dialog box remembers the last directory visited by the user when the application posts the dialog box again.

The following flags are available for the `opts` argument:

AFXSELECTFILE_EXISTING

Allows the selection of an existing file only.

AFXSELECTFILE_MULTIPLE

Allows the selection of multiple existing files only.

AFXSELECTFILE_DIRECTORY

Allows the selection of an existing directory only.

AFXSELECTFILE_REMOTE_HOST

Allows the opening of files on a remote host.

You specify the `patterns` argument as a series of patterns separated by `\n`. The value of the target specified by the `patternIndexTgt` argument determines which pattern is initially shown when the dialog box is posted.

The following is an example of how a file selection dialog box can be posted from a form:

```
def getFirstDialog(self):

    patterns = 'Output Database (*.odb)\nAll Files (*.*)'
    db = AFXFileSelectorDialog(self, 'Open ODB',
        self.nameKw, self.readOnlyKw, AFXSELECTFILE_EXISTING,
        patterns, self.patternIndexTgt)
    db.setReadOnlyPatterns('*.odb')
    self.setModal(True)
    return db
```

The following is an example of how a directory selection dialog box can be posted from another dialog box:

```
def onCmdDirectory(self, sender, sel, ptr):

    if not self.dirDb:
        self.dirDb = AFXFileSelectorDialog(self,
            'Select a Directory',self.form.dirNameKw,
             None, AFXSELECTFILE_DIRECTORY)
        self.dirDb.create()

    self.dirDb.showModal()
    return 1
```

Print dialog box

The **Print** dialog box provides standard printing functionality.

To post the **Print** dialog box from a button in your dialog box, you first access the print form mode by using the `getPrintForm` method of the FileToolsetGui class. This can be done by storing a pointer to the form as shown in the following example:

```
from sessionGui import FileToolsetGui

class MyMainWindow(AFXMainWindow):

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self, app, windowTitle=''):

        ...

        fileToolset = FileToolsetGui()
        self.printForm = fileToolset.getPrintForm()
        self.registerToolset(fileToolset,
            GUI_IN_MENUBAR|GUI_IN_TOOLBAR)

        ...
```

Then you can use the print form in your dialog box class, as shown below:

```
printForm = getAFXApp().getAFXMainWindow().printForm
FXButton(parent, 'Print...', None, printForm,
    AFXMode.ID_ACTIVATE)
```

To access the print form, you must construct and register the file toolset. However, you cannot access the print form from within a plug-in. As a result, you can only use the approach described here in a customized application.

Color selector dialog box

The AFXColorSelector widget provides the ability to choose a color from a predefined palette of colors.

This dialog box is posted by an AFXColorButton. For more information, see *Color buttons*.

Issuing Commands

This part describes how a dialog box can issue commands to the Abaqus/CAE kernel.

In this section:

- *Commands*
- *Modes*

Commands

This section describes the role of commands in the Abaqus GUI toolkit.

In this section:

- *An overview of commands*
- *The kernel and GUI processes*
- *Executing commands*
- *Kernel commands*
- *GUI commands*
- *AFXTargets*
- *Accessing kernel data from the GUI*
- *Receiving notification of kernel data changes*

An overview of commands

In Abaqus/CAE there are two types of commands: kernel commands and GUI commands.

Kernel commands

Kernel commands are used to build, analyze, and postprocess finite element models. Kernel commands are documented in the *Abaqus Scripting Reference Guide*.

GUI commands

GUI commands are used by the user interface to process input gathered from the user and to construct a kernel command string that is sent to the kernel for execution. GUI commands are documented in the *Abaqus GUI Toolkit Reference Guide*.

The kernel and GUI processes

Abaqus/CAE executes in two processes: a kernel process and a GUI process.

Kernel process

The kernel process holds all the data and methods that Abaqus/CAE uses to perform modeling operations; for example, creating parts and meshing the assembly. The kernel process can run independently of the GUI process.

GUI process

The GUI is a convenient way for the user to specify input to Abaqus/CAE. A kernel command string is sent from the GUI process to the kernel process via the inter-process communication (IPC) protocol. The kernel process interprets and executes the kernel command string. If the kernel command throws an exception, the exception is propagated back to the GUI process, where it should be caught and handled properly, typically by posting an error dialog box.

Abaqus/CAE uses an IPC protocol to achieve communication between the kernel and GUI processes. For example, the GUI often needs to query the kernel for a list of existing part names or for the values of a particular load that is about to be edited from a dialog box. Similarly, the GUI may need to be notified when some kernel value changes so that the GUI can update itself; for example, to post new job messages in the **Job Monitor** dialog box.

Abaqus/CAE uses targets and messages and the GUI updating process, built into the Abaqus GUI Toolkit, to achieve communication within the GUI process. For example, an options dialog box may need to update when the current viewport is changed or some widgets in a dialog box may need to be grayed out when the user clicks a particular button.

Figure 1 illustrates the communication between the kernel and the GUI processes when the user clicks on a button and then enters values in the dialog box that appears.

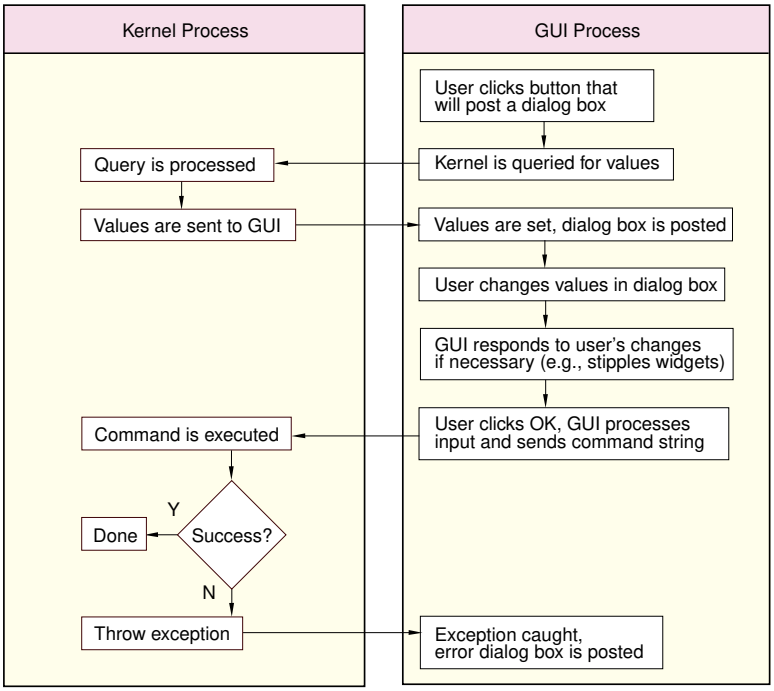


Figure 1: Communication between the kernel and GUI processes.

Executing commands

All commands are ultimately executed in the kernel process, but there are several ways this can be accomplished.

For example:

- You can execute kernel commands from a file by using the –start or –replay options on the command line.
- You can execute kernel commands from a file by using **File->Run Script**.
- You can type kernel commands in the Abaqus/CAE CLI.
- The GUI mode infrastructure can send a command string from the GUI to the kernel process for execution (see *Command processing* for details).
- You can issue a kernel command directly from the GUI using the `sendCommand` function.

The `sendCommand` function takes three arguments:

- A required string argument specifying the command to be executed in the kernel.
- Two optional Boolean arguments, `writeToReplay` and `writeToJournal`.

The optional Boolean arguments control whether or not the `sendCommand` function writes the command to the replay or journal file. By default, the `sendCommand` function writes the command to the replay file but not to the journal file. If the command modifies the model in any way, you should record the command in both the replay and journal files. However, if the command modifies only session data (such as the view of the viewport), you should record the command in the replay file, but you should not record it in the journal file. By convention, the user should be able to completely recreate the result of an interactive session by replaying its replay file. Only the commands that are written to the journal file will be available for data recovery in the event that the application exits.

Abaqus Scripting Interface commands automatically journal themselves. As a result, if you use the `sendCommand` function to issue an Abaqus Scripting Interface command, you should not set `writeToJournal`=True. Otherwise, the command will be recorded twice in the journal file. For more information, see *Abaqus/CAE command files*.

If you write your own kernel scripting module and functions, you should be aware that you can use the `journalMethodCall` function to record a command in the journal file. This option is preferable to using the *writeToJournal* argument in the `sendCommand` function. Your command should not call `journalMethodCall` if the command changes the Mdb object using built-in Abaqus Scripting Interface commands, because these are journaled by default. A command that changes the customData of the Mdb should call `journalMethodCall`. For an example that illustrates one common use of the `journalMethodCall` function, see *journalMethodCall*.

In general, you should enclose the `sendCommand` function in a try block to catch any exceptions that might be thrown by the kernel command. In order for exceptions to be caught, they should be class-based exceptions and not simply strings. For example:

```
from abaqusGui import sendCommand
try:
    sendCommand("mdb.customData.myCommand('Cmd-1', 50, 200)")
except ValueError as x:
    print('an exception was raised: ValueError: %s' % (x))
except:
    exc_type, exc_value = sys.exc_info()[:2]
    print('error. %s.%s'%(exc_type.__name__, exc_value))
```

Kernel commands

A kernel command can include a method, an object, and arguments.

A kernel command can consist of the following parts:

```
object + method + arguments (keywords)
```

Commands do not always have an object or even arguments, but they will always have a method.

For example:

```
session.viewports['Viewport: 1'].setValues(width=50, height=100)
|---------- object ------------|  method |---- arguments ----|

mdb.models[Model-1'].PointSection(name='Section-3', mass=1.0)
|----- object ------|-- method --|------- arguments --------|

session.viewports['Viewport: 1']. bringToFront()
|---------- object -------------|-- method --|

LeafFromElementSets(elementSets='PART-1-1.E1')
|----- method -----|------- arguments -------|
```

GUI commands

GUI commands are designed to work together with modes.
Modes perform the command processing and send the command to the kernel. For more information, see *Modes*.
This section describes how to construct and use GUI commands.

In this section:

- *Constructing GUI commands*
- *GUI commands and current objects*
- *Keeping the GUI and commands up-to-date*
- *Targets and messages*
- *Automatic GUI updating*
- *Data targets*
- *Option versus value mode*
- *AFXKeywords*
- *Expression evaluation*
- *Connecting keywords to widgets*
- *Boolean, integer, float, and string keyword examples*
- *Symbolic constant keyword examples*
- *Tuple keyword examples*
- *Table keyword example*
- *Object keyword example*
- *Defaults objects*

Constructing GUI commands

You use the AFXGuiCommand class to construct a GUI command.

The AFXGuiCommand class takes the following arguments:

mode

Modes are activated through a control in the GUI, typically a menu button. Once a mode is activated, it is responsible for gathering user input, processing the input, sending a command, and performing any error handling associated with the mode or the commands it sends. For a detailed discussion of modes, see *Modes*. The Abaqus GUI toolkit provides two modes:

Form modes

Form modes provide an interface to dialog boxes. Form modes gather input from the user using one or more dialog boxes.

Procedure modes

Procedure modes provide an interface that guides the user through a sequence of steps by prompting for input in the prompt area of the application.

method

A String specifying the method of the kernel command.

objectName

A String specifying the object of the kernel command.

registerQuery

A Boolean specifying whether or not to register a query on the object.

For example, the following statement creates a command to edit graphics options:

```
cmd = AFXGuiCommand(self, 'setValues',
    'session.graphicsOptions', True)
```

If you have more than one GUI command in a mode, the commands are processed in the same order in which they are created in the mode. For more examples of creating GUI commands, see *Form example* and *Procedure example*.

GUI commands and current objects

Most commands in Abaqus/CAE operate on the current object; for example, the current viewport or the current part.

As a convenience, modes recognize a special syntax when interpreting the object specified in a GUI command. If you place `%s` between square brackets following certain repositories, the mode replaces the `%s` with the current name. You should always use this `%s` syntax, as opposed to hard-coding a name, so that the current name will always be used in commands.

The following current objects are supported:

Object Specification	Mode Interpretation
`mdb.models[%s]`	Current model
`mdb.models[%s].parts[%s]`	Current part
`mdb.models[%s].sketches[%s]`	Current sketch
`session.odbs[%s]`	Current output database
`session.viewports[%s]`	Current viewport

Keeping the GUI and commands up-to-date

If a command edits an object, you should request that a query be registered on that object by specifying True for the `registerQuery` argument in the GUI command constructor. Registering a query will cause the keywords associated with the AFXGuiCommand to be updated with the kernel values when the mode is started and any time the kernel values change.

For example,

```
cmd = AFXGuiCommand(
    mode, 'PointSection', 'mdb.models[%s]', True)
```

In addition, modes recognize `session.viewports[%s]` as a special repository. The mode registers a query on the session automatically so that the command will be kept up-to-date if the user switches the current viewport. The following examples illustrate the special syntax:

```
cmd = AFXGuiCommand(
    mode,'setValues','session.viewports[%s]', True)

cmd = AFXGuiCommand(
    mode,'bringToFront','session.viewports[%s]', True)
```

Targets and messages

The Abaqus GUI Toolkit employs a target/message system to achieve communication within the GUI process.

The target/message system is in contrast to, for example, Motif's callback mechanism. All widgets can send and receive messages from any other widget. A message consists of two components:

- A message `type`
- A message ID

The message type indicates what kind of event occurred; for example, clicking a button. The message ID identifies the sender of the message.

Most widgets in the Abaqus GUI Toolkit take arguments that specify their target and their ID. Even if a widget does not take a target and ID as arguments, you can set these attributes using the `setTarget` and `setSelector` methods. For example,

```
FXButton(parent, 'Label', tgt=self, sel=self.ID_1)

groupBox = FXGroupBox(parent)
groupBox.setTarget(self)
groupBox.setSelector(self.ID_2)
```

Widgets are capable of sending several types of messages. Two of the most common message types are SEL_COMMAND and SEL_UPDATE. The SEL_COMMAND message type generally indicates that a widget was "committed"; for example, the user clicked a push button. The SEL_UPDATE message is sent when a widget is requesting its target to update its state; for more information, see *Automatic GUI updating*.

A message is routed to a message handler using a map defined in the target class. You add an entry in the map by specifying which method to call when a message of a certain type and ID is received. These concepts are illustrated in *Figure 1*.

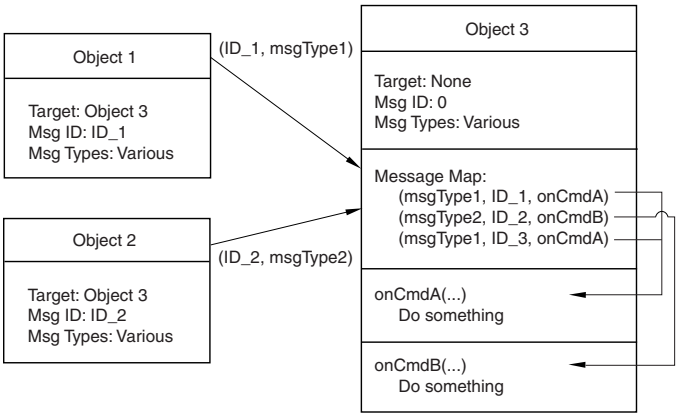


Figure 1: Targets and messages.

The message map is defined by using the FXMAPFUNC function (see example below). This macro takes four arguments: self, message type, message ID, and method name. The method name must be qualified by the class name: className.methodName. When a message is received whose type and ID match those defined in an FXMAPFUNC entry, the corresponding method will be called. If you have a large range of IDs that you want to define in the message map, you can use the FXMAPFUNCS function, which takes one additional argument: self, message type, start message ID, end message ID, and method name.

Objects react to messages using message handlers. All message handlers have the same prototype, which contains the following:

- The sender of the message
- The message selector
- Some “user data”

You can extract the type and ID of the message from the selector using the `SELTYPE` and `SELID` functions.

The following code shows how message maps, message IDs, and message handlers work together:

```
class MyClass(BaseClass):

    [
        ID_1,
        ID_2,
        ID_LAST
    ] = range(BaseClass.ID_LAST, BaseClass.ID_LAST+3)

     def __init__(self):

         BaseClass.__init__(self)
         FXMAPFUNC(self, SEL_COMMAND, self.ID_1,
             MyClass.onCmdPrintMsg)
         FXMAPFUNC(self, SEL_COMMAND, self.ID_2,
             MyClass.onCmdPrintMsg)

         FXButton(self, 'Button 1', None, self, self.ID_1)
         FXButton(self, 'Button 2', None, self, self.ID_2)

     def onCmdPrintMsg(self, sender, sel, ptr):

         if SELID(sel) == self.ID_1:
             print 'Button 1 was pressed.'
         elif SELID(sel) == self.ID_2:
             print 'Button 2 was pressed.'
         return 1
```

The previous example starts by generating a list of IDs for use in the derived class. Since a widget has a specific target, the ID of a widget does not have to be globally unique; it needs to be unique only within the target’s class and base classes. To handle this numbering automatically, the convention is to define ID_LAST in each class. A derived class should begin its numbering using the value of ID_LAST defined in its base class. In addition, a derived class should define its own ID_LAST as the last ID in the derived class. A class that derives from the derived class will then be able to make use of that ID to begin its numbering. ID_LAST should not be used by any widget. The only purpose of ID_LAST is to provide an automatic numbering scheme between classes.

The example continues by constructing a message map by adding entries using the `FXMAPFUNC` function. In this example, when a message of type `SEL_COMMAND` and an ID of `ID_1` or `ID_2` is received, the script calls the `onCmdPrintMsg` method.

The two button widgets have their target set to self (`MyClass`). However, when each widget sends a message, the widget sends a different message ID and the message handler checks the ID to determine who sent the message. For example, if the user clicks the first button, the button sends a `(ID_1, SEL_COMMAND)` message to `MyClass`. The class’s message map routes that message to the `onCmdPrintMsg` method. The `onCmdPrintMsg` method checks the ID of the incoming message and prints `Button 1 was pressed`.

It is important that your message handlers return the proper value to ensure that the GUI is kept up-to-date. Returning a `1` in a message handler tells the toolkit that the message was handled. In turn, if a message is handled, the toolkit assumes that something may have changed that requires an update, and the toolkit initiates a GUI update process. Returning a `0` in a message handler tells the toolkit that the message was not handled; therefore, the toolkit does not initiate a GUI update process.

Messages are normally sent by the GUI infrastructure as the result of some interaction in the GUI. However, you can send a message directly to an object by calling its `handle` method. The `handle` method takes three arguments: *sender*, *selector*, and *userData*. The sender is generally the object that is sending the message. The selector is made up of the message ID and the message type. You can use the `MKUINT` function to create a selector, for example, `MKUINT(ID_1, SEL_COMMAND)`. The user data must be *None* since this feature is not supported in the Abaqus GUI Toolkit.

Automatic GUI updating

GUI updating is initiated automatically by the Abaqus GUI Toolkit when there are no more events to be handled, usually when the GUI is idle and waiting for some user interaction.
During the automatic GUI update process, each widget sends a SEL_UPDATE message to its target asking to be updated. In this way the GUI is constantly polling the application state to keep itself up-to-date.

For example, during automatic GUI updating, a check button sends an update message to its target. The target checks some application state and determines whether or not the check button should be checked. If the button should be checked, the target sends back an ID_CHECK message; otherwise, it sends an ID_UNCHECK message.

Widgets in the toolkit are bidirectional; that is, they can be in either a *push* state or a *pull* state.

push state

In a *push* state the widgets are collecting and sending user input to the application. When a widget is in the *push* state, it does not participate in the automatic GUI updating process. Because the widget is not participating in the automatic GUI updating process, the user has control over the input, rather than the GUI attempting to update the widget.

pull state

In a *pull* state the widgets are interrogating the application to keep up-to-date.

Data targets

This section describes how the data targets work.
The following sections describe how the Abaqus GUI Toolkit has extended this concept to keywords that are used to construct commands sent to the kernel.

In a typical GUI application you will want to do the following:

1. Initialize the values in a dialog box.
2. Post the dialog box to allow the user to make changes.
3. Collect the changes from the dialog box.

In addition, you may want the dialog box to update its state if some application state is updated while the dialog box is posted. Data targets are designed to make these tasks easier for the GUI programmer.

A data target acts as a bidirectional intermediary between some application state and GUI widgets. More than one widget can be connected to a data target, but a data target acts on only one piece of application state. When the user uses the GUI to change a value, the application state monitored by the data target is updated automatically. Conversely, when the application state is updated, the widget connected to the data target is updated automatically.

As described in *Automatic GUI updating*, widgets can be in a push state or a pull state.

Push state

In a push state the widgets are collecting and sending user input to the application. *Figure 1* illustrates how a data target works with a widget that is in a push state. The sequence is as follows:

1. First, the user enters a value of `7` in the text field and then presses **Enter**.
2. This triggers the text field widget to send an `(ID, SEL_COMMAND)` message to its target—the data target.
3. The data target responds by sending the sender—the text field widget—a message requesting the value in the text field. The data target uses that value to update the value of its data.

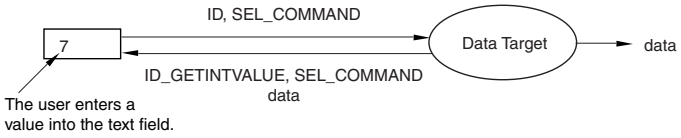


Figure 1: A data target with a text field widget in push state.

Pull state

In a pull state the widgets are interrogating the application to keep up-to-date. *Figure 2* illustrates how a data target works with a widget that is in a pull state. The sequence is as follows:

1. When the GUI is idle, it initiates a GUI update.
2. The GUI update triggers each widget to send an `(ID, SEL_UPDATE)` message to its target.
3. In this case the data target responds by sending the sender—the text field widget—a message telling it to set its value to the value of the data target's data.

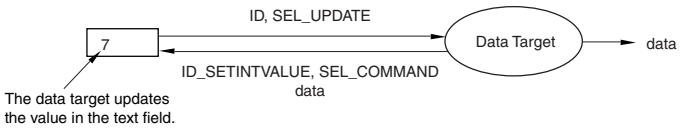


Figure 2: A data target with a text field widget in a pull state.

Option versus value mode

A data target works in one of two modes: value or option.
The examples in the previous section described the value mode. You use the value mode when the actual value of some data is of interest. In contrast, you use the option mode when you require a selection from many items and the value is not of particular importance.

For data targets operating in the option mode with a widget in the push state, the behavior is similar to the value mode described in the previous section. When the user clicks a button, the button sends an (ID, SEL_COMMAND) message to its target. In turn, the target responds by sending the sender a message requesting it to update the data target's data to the value of the sender's message ID.

For data targets operating in the option mode with a widget in the pull state, the behavior is slightly different from the value mode described in the previous section. During a GUI update the data target sends either a check or uncheck message back to the sender, depending on whether the sender's ID matches the value of the data target's data.

For example, *Figure 1* illustrates a data target operating in the option mode with three radio buttons in the pull state. Suppose that the value of the data being monitored by the data target is 13 and the message IDs of the radio buttons are 2, 13, and 58, respectively. The sequence is as follows:

1. During a GUI update the first radio button sends a (2, SEL_UPDATE) message to the data target.
2. The data target compares the message ID (2) to the value of its data (13) and sends an uncheck message back to the radio button since the values do not match.
3. The second radio button then sends a (13, SEL_UPDATE) message to the data target.
4. The data target compares the values and sends a check message back to the radio button since the values do match.
5. Similarly, the third button receives an uncheck message from the data target since the values of the message ID and its data do not match.

In this way the Abaqus GUI Toolkit automatically maintains the radio button behavior (only one button at a time will ever be checked).

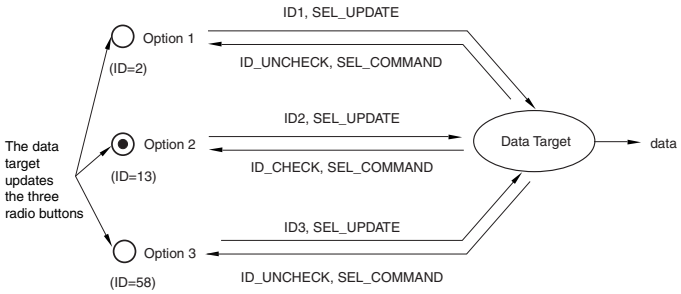


Figure 1: A data target operating on three radio buttons in option mode and a pull state.

AFXKeywords

Keywords generate the arguments to a GUI command. These keywords belong to the command, but the keywords are also stored as members of the mode.
As a result, you can easily connect the keywords to widgets in a dialog box that updates the value of the keywords. For more information, see *Dialog boxes*.

AFXKeyword is the base class for keywords in the toolkit. The AFXKeyword class derives from a data target, so it automatically keeps the GUI and application data synchronized with each other. For more information, see *Data targets*.

The AFXKeyword class extends the functionality of the FXDataTarget class by holding additional values, such as the name of the keyword, a default value, and a previous value. The keyword's GUI command uses this information to construct a kernel command string.

You can designate a keyword as optional or required. A required keyword is always issued by the GUI command. An optional keyword, whose values have not changed since the last commit of the command, is not issued by the GUI command. If none of the keywords has changed since the last commit, no GUI command will be issued when the mode is committed.

The following types of keywords are supported:

- `AFXIntKeyword(cmd, name, isRequired, defaultValue)`
- `AFXFloatKeyword(cmd, name, isRequired, defaultValue)`
- `AFXStringKeyword(cmd, name, isRequired, defaultValue)`
- ```
  AFXBoolKeyword(cmd, name, booleanType, isRequired,
      defaultValue)
  ```
- `AFXSymConstKeyword(cmd, name, isRequired, defaultValue)`
- ```
  AFXTupleKeyword(cmd, name, isRequired, minLength,
      maxLength, opts)
  ```
- ```
  AFXTableKeyword(cmd, name, isRequired, minLength,
      maxLength, opts)
  ```
- `AFXObjectKeyword(cmd, name, isRequired, defaultValue)`

The type of data supported by each keyword is implied from the name of its constructor, except for AFXObjectKeyword. An object keyword is one that supports specifying a variable name as the keyword's value.

The prototypes for all the keywords are similar. The first two arguments of a keyword are:

- A GUI command object.
- A String specifying the name of the keyword.

All keywords also support an argument that determines whether the keyword is required or optional. If a keyword is required, it will always be sent with the command. If a keyword is optional, it will be sent only if its value changes. However, if the keyword is connected to a widget that is hidden, then the keyword will not be sent regardless of whether it is required or optional.

Most keywords support the specification of a default value. When you construct a keyword, its value is set to the default value. If you use the keyword's `setDefaultValue` method to change the default value, you will not affect the value of the keyword unless you also call the keyword's `setValueToDefault` method. In contrast, if you want to change only the value of the keyword, without changing its default value, you should use the keywords' `setValue` method.
  ```

When the mode issues the command to the kernel, the keywords will be ordered in the same order in which they were created in the mode.

When storing keywords in the mode class, the convention is to name the keyword object using the same name as the keyword label plus `Kw`. For example,

```
self.rKw = AFXIntKeyword(self.cmd, 'r',  True)
self.tKw = AFXFloatKeyword(self.cmd, 't',  True)
self.nameKw = AFXStringKeyword(cmd, 'name', True, 'Part-1')
self.twistKw = AFXBoolKeyword(cmd, 'twist',
    AFXBoolKeyword.ON_OFF, 0)
self.typeKw = AFXSymConstKeyword(cmd, 'type', True,
    SHADED.getId())
self.imageSizeKw = AFXTupleKeyword(cmd, 'imageSize', False,
    1, 2, AFXTUPLE_TYPE_FLOAT)
```

Expression evaluation

The AFXFloatKeyword and AFXIntKeyword both support expression evaluation.

You can type a numeric expression into a text field connected to an AFXFloatKeyword or AFXIntKeyword and that expression will get evaluated. For example, you could type any of the following expressions into a text field connected to an AFXFloatKeyword:

```
3 + (7 * 22)
2 * 3.1415 * 1.5
125/55.8
```

The expression will get sent in the command, so it will appear in the replay and journal files, but once the command is processed in the kernel, only the resultant value gets stored and the expression is lost.

Expression evaluation is always available with an AFXFloatKeyword, but it is optional for AFXIntKeyword (the default is to perform expression evaluation). If you connect an AFXIntKeyword to an AFXList or AFXComboBox and the choices shown in the list or combo box do not represent numeric values, you must disable expression evaluation. For example:

```
Form code snippet:

    self.orderKw = AFXIntKeyword(cmd=cmd, name='order',
        isRequired=False, defaultValue=1, evalExpression=False)

Dialog code snippet:

    combo = AFXComboBox(self, 8, 3, 'Order:', form.orderKw, 0)
    combo.appendItem('First', 1)
    combo.appendItem('Second', 2)
    combo.appendItem('Third', 3)
```

The command snippet from this code will look like:

```
someCommand(order=2, ...)
```

Connecting keywords to widgets

Keywords are used in the GUI by setting them as the targets of widgets.

The AFXDataDialog class takes a mode as one of its constructor arguments. The dialog box uses the mode provided in the constructor to access the keywords stored in the mode. In addition, the dialog box uses the keywords as targets of widgets in the dialog.

In addition to a target, a widget also has a message ID. It is important that the appropriate ID be set for the keyword to operate in the proper mode: value or option. For more information, see *Data targets*. In most cases a value of zero should be used for the message ID; a value of zero indicates that the keyword should operate in value mode. The table below summarizes the message ID usage with keywords, and the following sections give examples for each type of keyword.

Keyword	ID	Description
`AFXIntKeyword`	0	Keyword operates in value mode. Use this when the keyword is connected to a text field, list, combo box, or spinner.
	>0	Keyword operates in option mode. Use this when the keyword is connected to a radio button.
`AFXFloatKeyword`	0	Keyword operates in value mode.
`AFXStringKeyword`	0	Keyword operates in value mode.
`AFXBoolKeyword`	0	Keyword operates in value mode. This keyword should be used only with widgets that allow only Boolean values, such as `FXCheckButton`.
`AFXSymConstKeyword`	0	Keyword operates in value mode. Use this value when the keyword is connected to a list or combo box.
	> 0	Keyword operates in option mode. Use the value of the Symbolic Constant's ID when the keyword is connected to a radio button. Do not use this keyword with `FXCheckButton`.
`AFXTupleKeyword`	0	Keyword operates in value mode. Use this value when the entire tuple is gathered from a single widget.
	1, 2, 3,.	Keyword operates in value mode for only the *n*th element of the tuple, where *n*=ID. Use this value when the input for each element is gathered from separate widgets.
`AFXTableKeyword`	0	Keyword operates in value mode.
`AFXObjectKeyword`	0	Keyword operates in value mode.

Boolean, integer, float, and string keyword examples

The following statements illustrate the use of Boolean, integer, float, and string keywords.

```
# Boolean keyword with a checkbox
#
FXCheckButton(self, 'Show node labels', mode.nodeLabelsKw, 0)

#Boolean keyword with option tree list
#
self.tree = AFXOptionTreeList(parent, 6)
self.treeitem.addItemLast('Item 1', mode.item1Kw)

# Integer keyword
#
AFXTextField(self, 8, 'Number of CPUs:', mode.cpusKw, 0)

combo = AFXComboBox(self, 8, 3, 'Number:', mode.numberKw, 0)
combo.appendItem('1', 1)
combo.appendItem('2', 2)
combo.appendItem('3', 3)

# Float keyword
#
AFXTextField(self, 8, 'Radius:', mode.radiusKw, 0)

# String keyword
#
AFXTextField(self, 8, 'Name:', mode.nameKw, 0)
```

Symbolic constant keyword examples

Symbolic constants provide a way to specify choices for a command argument that make the command more readable. For example, there are three choices for the *renderStyle* argument in display options commands.

We could number these choices using integer values from 1 to 3. However, using integer values would result in a command that is not very readable; for example, `renderStyle=2`. Alternatively, if we define symbolic constants for each choice, the command becomes more readable; for example, `renderStyle=HIDDEN`. Internally, symbolic constants contain an integer ID that can be accessed via its `getId()` method. Symbolic constants can be used in both the GUI and kernel processes. Typically you should create a module that defines your symbolic constants and then import that module into both your kernel and GUI scripts.

You can import the SymbolicConstant constructor from the symbolicConstants module. The constructor takes a single string argument. By convention, the string argument uses all capital letters, with an underscore between words, and the variable name is the same as the string argument. For example,

```
from symbolicConstants import SymbolicConstant
AS_IS = SymbolicConstant('AS_IS')
```

In the case of symbolic constant keywords, you can use a value of zero or the value of the ID of a symbolic constant for the message ID. Symbolic constants have a unique integer ID that is used to set the value of symbolic constant keywords along with a string representation that is used in the generation of the command. To access the integer ID of a symbolic constant, use its `getId` method.

If the keyword is connected to a list or combo box widget, you should use a value of zero for the ID in the widget constructor. The AFXList, AFXComboBox, and AFXListBox widgets have been designed to handle symbolic constant keywords as targets. When items are added to a list or combo box, a symbolic constant's ID is passed in as user data. These widgets react by setting their value to the item whose user data matches the value of their target, as opposed to setting their value to the item whose index matches the target's value. The following example illustrates how a combo box can be connected to a symbolic constant keyword:

```
combo = AFXComboBox(hwGb, 18, 4, 'Highlight method:',
    mode.highlightMethodHintKw, 0)
combo.appendItem('Hardware Overlay', HARDWARE_OVERLAY.getId())
combo.appendItem('Software Overlay', SOFTWARE_OVERLAY.getId())
combo.appendItem('XOR', XOR.getId())
combo.appendItem('Blend', BLEND.getId())
```

If the keyword is connected to a radio button, you should use the ID of the symbolic constant that corresponds to that radio button for the message ID. Since the ID of all symbolic constants is greater than zero, this tells the keyword to operate in option mode. The following example illustrates how symbolic constant keywords can be used with radio buttons:

```
from abaqusConstants import *
...

# Modeling Space
#
gb = FXGroupBox(self, 'Modeling Space',
    FRAME_GROOVE|LAYOUT_FILL_X)
FXRadioButton(gb, '3D', mode.dimensionalityKw,
    THREE_D.getId(), LAYOUT_SIDE_LEFT)
FXRadioButton(gb, '2D Planar', mode.dimensionalityKw,
    TWO_D_PLANAR.getId(), LAYOUT_SIDE_LEFT)
FXRadioButton(gb, 'Axisymmetric',
    mode.dimensionalityKw, AXISYMMETRIC.getId(),
    LAYOUT_SIDE_LEFT)
```

Tuple keyword examples

In the case of tuple keywords, a value of zero for the message ID indicates that the entire tuple will be updated.

For example, you can use a single text field to collect *X*-, *Y*-, and *Z*-inputs from the user. In this case the comma-separated string entered by the user is used to set the entire value of the tuple keyword. For example, if you define a tuple keyword as follows:

```
self.viewVectorKw = AFXTupleKeyword(cmd, 'viewVector',
    True, 3, 3)
```

you can connect the tuple keyword to a single text field as follows:

```
AFXTextField(self, 12, 'View Vector (X,Y,Z)',
    mode.viewVectorKw, 0)
```

Alternatively, you can use three separate text fields to collect *X*-, *Y*-, and *Z*-inputs. Each of the text field widgets uses a message ID equal to the element number (1-based) of the tuple to which they correspond. For example, 1 corresponds to the first element of the tuple; 2 corresponds to the second element in the tuple, etc. In this case we can connect the keyword to three text fields as follows:

```
AFXTextField(self, 4, 'X:', mode.viewVectorKw, 1)
AFXTextField(self, 4, 'Y:', mode.viewVectorKw, 2)
AFXTextField(self, 4, 'Z:', mode.viewVectorKw, 3)
```

Table keyword example

The AFXTableKeyword must be connected to a table widget. This type of keyword will result in a command argument that is a tuple of tuples. The values in a table keyword can be Ints, Floats, or Strings.

The default minimum number of rows is 0, and the default maximum number rows is –1, indicating that the number of rows is unlimited. Tables can vary in size because the user can add or delete rows; as a result, you usually specify the defaults for the minimum and maximum number of rows. For example, to generate a command that creates XY data, you can define the following keywords in the form

```
self.cmd = AFXGuiCommand(self, 'XYData', 'session')
self.nameKw = AFXStringKeyword(self.cmd, 'name', True)
self.dataKw = AFXTableKeyword(
   self.cmd, 'data', True, 0, -1, AFXTABLE_TYPE_FLOAT)
```

In the dialog box you connect the table keyword to a table using a selector value of zero.

```
table = AFXTable(vf, 6, 3, 6, 3,
   form.dataKw, 0, AFXTABLE_NORMAL|AFXTABLE_EDITABLE)
```

If you have a table in which you are interested in the values of only a single column, you can make use of the AFXColumnItems object to track selections. For example, if a table contains Name and Description columns, you might only need the names in the selected rows for your command. In that case you could use AFXColumnItems to keep a tuple keyword up to date with the names in the selected rows of the table as shown in the following code:

```
ci = AFXColumnItems(referenceColumn=0, tgt=form.tupleKw, sel=0)
table = AFXTable(self, 4, 2, 4, 2, ci, 0,
   AFXTABLE_NORMAL|AFXTABLE_ROW_MODE|AFXTABLE_EXTENDED_SELECT)
```

Object keyword example

The AFXObjectKeyword has a variable name for its value.

In most cases you use an AFXObjectKeyword in a command that is preceded by some setup commands. For example,

```
p = mdb.models['Model-1'].parts['Part-1']
session.viewports['Viewport: 1'].setValues(displayedObject=p)
```

In this example, in the form you would issue the first command “manually,” and use an object keyword as part of an AFXGuiCommand to have the second command issued using “p” as the variable name. For example,

```
self.cmd = AFXGuiCommand(self, 'setValues',
    'session.viewports[%s]')
self.doKw = AFXObjectKeyword(self.cmd, 'displayedObject',
    True, 'p')
```

You also use an AFXObjectKeyword in procedures that require picking. For more information, see *Picking in procedure modes*.

Defaults objects

A defaults object can be used to restore the values of the keywords in a command to their default values when the user presses the **Defaults** button in the dialog box.

You can register a defaults object with a command as follows:

```
self.registerDefaultsObject(cmd,
    'session.defaultGraphicsOptions')
```

In addition, the AFXGuiCommand class has a `setKeywordValuesToDefaults` method that you can use to initialize the state of all keywords in a command. In most cases you use the `setKeywordValuesToDefaults` method to initialize the state of all keywords in the `getFirstDialog` method of the mode. As a result, the application will initialize the value of the keywords in a command each time the dialog box is posted.

If no defaults object is specified, the command uses the default values specified in the keyword's constructor when the user presses the **Defaults** button in the dialog box.

AFXTargets

Targets are similar to keywords in that they automatically keep their data synchronized with the GUI; however, targets do not participate in command processing.

Targets are typically used to monitor the value of some widget in the GUI that is not directly related to a command; for example, selecting the type of load to create from the **Create Load** dialog box. The following types of targets are supported:

- AFXIntTarget(initialValue)
- AFXFloatTarget(initialValue)
- AFXStringTarget(initialValue)

Accessing kernel data from the GUI

You can use the `abaqusGui` module or the `kernelAccess` module to access the kernel mdb and session objects from the GUI in Abaqus/CAE.

Each module has advantages and disadvantages for programming in the GUI. You access the objects from each module in the same way:

```
from abaqusGui import mdb, session
```

or

```
from kernelAccess import mdb, session
```

In each case the imported objects are proxies for the actual objects in the kernel.

You can query the `abaqusGui` module mdb and session proxy objects for attributes of objects, but they cannot be used for arbitrary method calls (repository methods such as `keys()`, `values()`, and `items()` are allowed). The `abaqusGui` proxy objects are regularly updated from the kernel, and accessing them is an in-process function call (fast). However, in some cases the proxy objects can get out of date. For example, when a script is running the proxy objects are not updated until it ends.

You can use the `kernelAccess` module mdb and session proxy objects to execute any Abaqus Scripting Interface kernel command. In addition to querying attributes of the kernel objects, you can call their methods and obtain any return values as if you were executing the code in the kernel. The `kernelAccess` proxy objects are always up-to-date because accessing them calls the kernel object synchronously, creating inter-process communication (IPC) traffic. This immediate interaction with the kernel creates a performance disadvantage when you use the `kernelAccess` proxy objects instead of the `abaqusGui` module proxy objects. For example, call the `getVolume` method of the Part object:

```
from kernelAccess import mdb, session
partNames = mdb.models['Model-1'].parts.keys()
v = mdb.models['Model-1'].parts['Part-1'].getVolume()
```

This procedure involves GUI-kernel communication via the IPC mechanism, so it is not recommended for use where performance is a concern. In other words, you should only use this procedure for accessing data or calling methods that do not take a “long time” to execute. If performance does become a problem, you can access the mdb and session objects from the `abaqusGui` module instead of the `kernelAccess` module.

Although you can import the `kernelAccess` module in a script that is executed before the application startup script has completed, you cannot query the mdb and session objects until the application startup script has completed. In other words, you can import the `kernelAccess` module in your scripts in code that is executed during the initial construction of the GUI; however, you should not attempt to access either the mdb or session object until it is needed because of some user interaction in the GUI. For more information, see *Startup script*

Receiving notification of kernel data changes

This section describes how the GUI can be notified when kernel objects and custom kernel objects are modified outside the GUI process.

In this section:

- *Automatically registering a query on kernel objects*
- *Manually registering a query on kernel objects*
- *Using registerQuery on kernelAccess proxy objects*
- *Recognizing when custom kernel data change*

Automatically registering a query on kernel objects

This section describes automatically registering a query on kernel object.

Queries provide a mechanism that allows the GUI process to be notified when data in the kernel change. *Keeping the GUI and commands up-to-date* describes how you use the *registerQuery* argument of the `AFXGuiCommand` constructor. The *registerQuery* argument is a Boolean flag that specifies whether to register a query automatically on the object being edited by the specified kernel command. If the kernel object specified in the `AFXGuiCommand` constructor changes, the infrastructure updates the keywords in the GUI with the latest values. As a result, you do not need to register a query explicitly. By default, *registerQuery*= False, and the query is not automatically registered.

For example,

```
cmd = AFXGuiCommand(mode,'setValues', mdb.models[%s].parts[%s],
    True)
```

In this example, if the user changes the current part, the path to the `setValues` method is updated to reflect the new current part. As a result, when the user clicks **OK** to commit a customized dialog box, the mode issues a `setValues` command that modifies the current part.

Manually registering a query on kernel objects

For objects not directly related to a command, such as a repository, you may wish to register a query yourself. You can register a query on an Abaqus/CAE object using the `registerQuery` method.

The `registerQuery` method takes a callback function as an argument. When the object upon which the query is registered changes, the infrastructure automatically calls the function supplied in the `registerQuery` method. For example,

```
from abaqusGui import *

def onPartsChanged():
    print('The parts repository changed.')
    keys = mdb.models['Model-1'].parts.keys()
    print('The new keys are:', keys)

mdb.models['Model-1'].parts.registerQuery(onPartsChanged)
```

In the previous example, if a part is created, deleted, renamed, or edited, the `onPartsChanged` method will be called.

The `registerQuery` method takes an optional second argument that determines whether or not the callback is called when the query is first registered. By default, this argument is True, and the callback will be called when the query is first registered. If you specify False as the second argument, the query callback is not called when the query is first registered. Delaying the query callback can prevent errors in certain situations; for more information, see *Using registerQuery on kernelAccess proxy objects*.

Since registered queries create “traffic” between the kernel and GUI processes, you should unregister queries when you do not need them. To unregister a query, use the `unregisterQuery` method and pass the same arguments that you used in the `registerQuery` method. In most cases, you register queries within the `show` method that you write for your dialog box that needs the queries. Similarly, you unregister queries within the `hide` method that you write for your dialog box. If you do not unregister a query and the query fires when the dialog box is not posted, the application may exit if the callback tries to modify a widget in the dialog box.

If the user creates, deletes, renames, or edits a part in the following example, the application will call the `onPartsChanged` method and update the dialog box:

```
class MyDialog(AFXDataDialog):

    ...

    def onPartsChanged(self):

        # Code to update the part list
        # in the dialog box

    def show(self):

        from kernelAccess import mdb
        mdb.models['Model-1'].parts.registerQuery(
            self.onPartsChanged)
        AFXDataDialog.show(self)

    def hide(self):

        from kernelAccess import mdb
        mdb.models['Model-1'].parts.unregisterQuery(
            self.onPartsChanged)
        AFXDataDialog.hide(self)
```

Using registerQuery on kernelAccess proxy objects

It is possible to call the registerQuery method on a kernelAccess module proxy object instead of the abaqusGui proxy object. However, internally the query is always registered on the abaqusGui proxy object.

The two kinds of proxy objects are not always perfectly synchronized. In most cases, this will not matter. However, it can cause a problem if the query is registered while a change is being made on the kernel. For example,

```
from kernelAccess import mdb

def onPartsChanged():
    print('The parts repository changed.')
    keys = mdb.models['Model-1'].parts.keys() # OK
    print('The new keys are:', keys)
    if keys:
      mdb.models['Model-1'].parts[keys[0]].registerQuery(onPartsChanged) # Not
 OK
       # Internally the registerQuery method will be called on abaqusGui.mdb...

mdb.models['Model-1'].parts.registerQuery(onPartsChanged)
```

If a changeKey command that affects the names in the parts repository is subsequently issued, the above example will fail. The keys (part names) are obtained using the kernelAccess mdb proxy object and contain the changed name. However, the registerQuery method—based on the abaqusGui proxy object—does not see the new names until the changeKey command is completed. The registerQuery on the newly named part object (using keys()[0]) is being called within the callback, before the changeKey command is completed, and the callback is using the new name. Since the abaqusGui proxy for the parts repository has not yet been updated, a Keyerror is raised.

Run the above example in the GUI, create a part, then enter the following in the command line interface (CLI):

```
>>> mdb.models['Model-1'].parts.changeKey('Part-1', 'ROD')
```

You will get the following error:

```
Traceback (most recent call last):
 File "path and filename of the example script",
line 9, in onPartsChanged
    mdb.models['Model-1].parts[partNames[0]].registerQuery
(onPartsChanged) # Not OK
KeyError: 'ROD'
```

The error can cause Abaqus/CAE to stop responding. Setting the second argument on registerQuery to False prevents the callback from being called immediately and prevents this potential error.

Recognizing when custom kernel data change

To receive notification in the GUI of changes made to custom kernel objects, those kernel objects must make use of special classes provided by the `customKernel` module.

The `customKernel` module provides the following special classes, all of which are capable of notifying the GUI when the contents of the class changes:

- `CommandRegister` allows you to create general classes. For more information, see *CommandRegister class*.
- `RepositorySupport` allows you to create repositories below other repositories. For more information, see *RepositorySupport*.
- `RegisteredDictionary` allows you to create custom dictionaries. For more information, see *Registered dictionaries*.
- `RegisteredList` allows you to create custom lists. For more information, see *Registered lists*.
- `RegisteredTuple` allows you to create custom tuples. For more information, see *Registered tuples*.

For more information on the `customKernel` module, see *Extending the Abaqus Scripting Interface*.

Modes

A mode is a mechanism for gathering input from the user, processing that input, and then issuing a command to the kernel. This section describes the modes that are available in the Abaqus GUI Toolkit.

In this section:

- *About modes*
- *Mode processing*
- *Form modes*
- *Procedure modes*
- *Picking in procedure modes*

About modes

There are two types of modes: form modes and procedure modes.

Form Modes

Form modes provide an interface to standalone dialog boxes.

Procedure Modes

Procedure modes provide an interface that uses the prompt area to guide the user through a sequence of steps that collect input from dialog boxes or from selections in the viewport.

If a mode needs to perform drawing or highlighting in the current viewport, the mode must be a procedure mode. Because Abaqus/CAE highlights objects that the user picks, any mode that requires the user to pick in the viewport must also be a procedure mode. Procedure modes ensure that only one procedure at a time has control over the scene in the current viewport. If two different procedures could highlight different portions of the model for different purposes, the resulting display would be confusing and ambiguous.

Mode processing

Modes are typically activated by a button in the GUI. Once a mode is activated, it is responsible for gathering user input, processing the input, sending a command, and performing any error handling associated with the mode or the commands it sends. This section describes how modes are processed.

In this section:

- *The mode processing sequence*
- *Activating a mode*
- *Step and dialog box processing*
- *Command processing*
- *Work in progress*
- *Command error handling*

The mode processing sequence

During the input gathering process, the mode allows you to perform some intermediate error checking. For example, if the user is supposed to enter a value between zero and one but enters a value outside this range, you can flag the error before continuing to collect more input. After all the inputs are collected from the user, the mode verifies the input, constructs the command, and sends the command to the kernel. If there is an exception thrown by the kernel, the mode will handle the exception.

The mode processing sequence is shown in *Figure 1*.

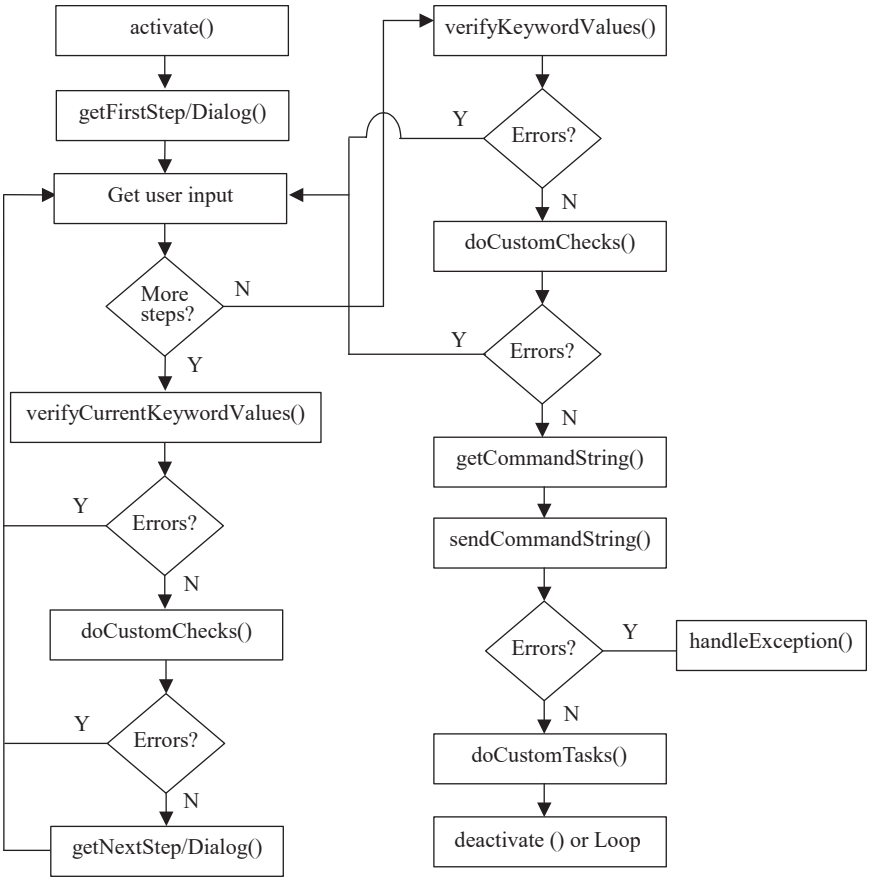


Figure 1: The mode processing sequence.

To provide custom processing in your mode, you can overwrite many of the methods shown in *Figure 1*. If you overwrite a method, you should use the exact same prototype for your method, including the same default values that the method may have. Refer to the *Abaqus GUI Toolkit Reference Guide* to determine the prototype of a method.

Activating a mode

A mode is usually activated by sending it a message with its ID set to ID_ACTIVATE and a type of SEL_COMMAND. This message causes the activate method of the mode to be called.
For more information, see *Targets and messages*.

If you need to do any processing before a mode begins to collect input from the user, you can redefine the activate method. For example, you can check that the current viewport contains a part before beginning a mode that requires the user to pick something on a part, as shown in the following method:

```
    def activate(self):

        if getDisplayedObjectType() == PART:
            AFXForm.activate(self)
        else:
            showAFXErrorDialog(getAFXApp().getAFXMainWindow(),
                 'A part must be displayed in the \
                  current viewport.')
```

If you write your own activate (or deactivate) method, you must call the base class version of that method if no error conditions are encountered. The base class methods perform additional processing necessary to make the mode function properly.

Step and dialog box processing

After a mode is activated, it cycles through a sequence of events collecting inputs from the user and verifying the inputs.

After the user commits each step or dialog box, the mode calls the following methods:

verifyCurrentKeywordValues

The `verifyCurrentKeywordValues` method calls the `verify` method for each keyword associated with the current step or dialog box, and the method posts an error dialog box if necessary. The `verifyCurrentKeywordValues` method returns True if no errors were encountered; otherwise, it returns False and terminates further processing.

doCustomChecks

The `doCustomChecks` method has an empty implementation in the base class. You can redefine this method to perform any additional checking of keyword values, typically to perform range checking or to check some interdependency between values. The `doCustomChecks` method should return True if no errors were encountered; otherwise, it should return False so that further command processing will be terminated. The `doCustomChecks` method is called by the mode during step and dialog box processing and during command processing.

Command processing

When the mode finishes collecting inputs from the user, it calls a series of methods. If needed, you can redefine some of the methods to customize the behavior of the mode.

The following list describes each of the methods called by the mode:

verifyKeywordValues

The `verifyKeywordValues` method calls the `verify` method for each keyword of each command associated with the mode and posts an error dialog box if necessary. The `verifyKeywordValues` method returns True if no errors were encountered; otherwise, it returns False and terminates further command processing.

doCustomChecks

The `doCustomChecks` method has an empty implementation in the base class. You can redefine this method to perform any additional checking of keyword values, typically to perform range checking or to check some interdependency between values. The `doCustomChecks` method should return True if no errors were encountered; otherwise, it should return False so that further command processing will be terminated. The `doCustomChecks` method is called by the mode during step and dialog box processing and during command processing.

The following example shows how you can use the `doCustomChecks` method to find an invalid value and, in response, to post an error dialog box and put the cursor into the appropriate widget. If the keyword is connected to a text field in the dialog, the `onKeywordError` method finds the text field widget, select its contents, and places the focus on the widget.

```
def doCustomChecks(self):

    if self.lengthKw.getValue() >= 1000:
        showAFXErrorDialog(self.getCurrentDialog(),
            'Length must be less than 1000.')
        self.getCurrentDialog().onKeywordError(self.lengthKw)
        return False
```

issueCommands

The `issueCommands` method is responsible for constructing the command string, issuing it to the kernel, handling any exceptions from the command, and deactivating the mode if necessary. The `issueCommands` method calls the following methods:

- `getCommandString`: This method returns a string that represents the commands collected from each command associated with the mode. Required keywords are always sent with the command, but optional keywords are sent only if their value has changed. The commands are issued to the kernel in the same order as the commands were constructed in the mode. If your command does not fit the standard style of the command generated by the mode, you can redefine this method to generate your own command string.

- `sendCommandString`: This method takes the command string returned from the `getCommandString` method and sends it to the kernel for processing. You should not overwrite this method or your mode may not perform properly.

- `doCustomTasks`: This method has an empty implementation in the base class. You can redefine this method to perform any additional tasks required after a command is processed by the kernel.

After calling these methods, the `issueCommands` method will deactivate the mode if the user pressed the **OK** button.

The `issueCommands` method also controls the writing of the command to the replay and journal files. The GUI infrastructure always calls `issueCommands` with `writeToReplay`=**True** and `writeToJournal`=**False**. If you want to change the behavior, you can overwrite this method and specify different values for the arguments. If you overwrite the `issueCommands` method you must specify both arguments, and you should always call the base class method from your method or your mode may not perform properly. For example:

```
def issueCommands(self, writeToReplay, writeToJournal):
    AFXForm.issueCommands(self, writeToReplay=True,
        writeToJournal=True)
```

In most cases, you do not need to call `issueCommands` since the infrastructure will call it automatically; however, if you interrupt the normal flow of mode processing, you must call `issueCommands` to complete the processing. For example, if before issuing a command you want to ask the user for permission to execute the command, you can post a warning dialog box from the `doCustomChecks` method. In this example you must return **False** from the `doCustomChecks` method to stop the command processing. The application will then wait for the user to make a selection from the warning dialog box. When the user clicks a button in the warning dialog box, you must catch the message sent by the dialog box to your form. If the user clicks **Yes**, you should continue the command processing as shown in the following example:

```
class MyForm(AFXForm):

    ID_OVERWRITE = AFXForm.ID_LAST

    def __init__(self, owner):

        AFXForm.__init__(self, owner)
        FXMAPFUNC(self, SEL_COMMAND,
            self.ID_OVERWRITE, MyForm.onCmdOverwrite)
        ...

    def doCustomChecks(self):

        import os
        if os.path.exists(self.fileNameKw.getValue()):
            db = self.getCurrentDialog()
            showAFXWarningDialog(db,
                'File already exists.\n\nOK to overwrite?',
                AFXDialog.YES|AFXDialog.NO, self,
                self.ID_OVERWRITE)
            return False

        return True

    def onCmdOverwrite(self, sender, sel, ptr):

        if sender.getPressedButtonId() == \
            AFXDialog.ID_CLICKED_YES:
                self.issueCommands(writeToReplay=True,
                    writeToJournal=True)
        return 1
```

Normally the GUI infrastructure takes care of sending commands to the kernel automatically when the mode is committed. If you need to issue a command before the mode is committed, you can call `issueCommands` yourself. In other cases you may want to send a command without using the form's infrastructure. You can send a command string directly to the kernel using the `sendCommand(cmd)` method. For more information, see *Executing commands*.

deactivate

After the mode has issued its commands successfully, it will call the `deactivate` method to perform various cleanup tasks, unless the mode loops or an “Apply” button was pressed, in which case the mode returns to wait for further input from the user. If you need to perform your own cleanup tasks for your mode, you can overwrite this method; but you should be sure to call the base class method as well to ensure that the mode is terminated properly as shown in the following example.

```
def deactivate(self):

    # Do your processing here

    # Call the base class method
    AFXForm.deactivate(self)
```

cancel

If you need to cancel a mode programmatically, as opposed to the user clicking on a **Cancel** button, you can call the mode's `cancel` method, taking the default values for its arguments. The `cancel` method will call the `deactivate` method, so the mode's cleanup tasks will still be performed.

If you want to give the user a chance to confirm whether a mode should be cancelled, you can have a bailout dialog invoked. If you are writing a form mode, you can specify the bailout flag in the constructor of your dialog box. If you are writing a procedure mode, you should write the `checkCancel` method. The return value of the `checkCancel` method determines if the user will be prompted for confirmation when the procedure is cancelled. For example:

```
def checkCancel(self):

    if self.getCurrentStep() == self.step1:
        # If cancelled in the first step, do  not
        # ask the user for confirmation to cancel.
        return AFXProcedure.BAILOUT_OK
    else:
        # After the first step, ask the user for
        # confirmation before cancelling.
        return AFXProcedure.BAILOUT_NOTOK
```

By default when the context changes in Abaqus/CAE, all forms are cancelled; for example, when the user opens a new database or changes the current model. If you have a form mode that you do not want to be cancelled, you can overwrite the base class implementation in your form code as follows:

```
def okToCancel(self):
    return False
```

Work in progress

If the command sent to the kernel takes more than a certain amount of time (approximately one second), the GUI will lock and the busy cursor will be displayed. If you want to provide additional feedback about the progress of your command, you can add work-in-progress commands to your kernel code.
For more information, see *Status commands*.

The following statements illustrate how you can use the `milestone` command to provide feedback on the progress of a volume computation:

```
numObjects = 4
for i in range(numObjects+1):
    milestone('Computing total volume', 'parts', i, numObjects)
    ...
    compute volume here
    ...
```

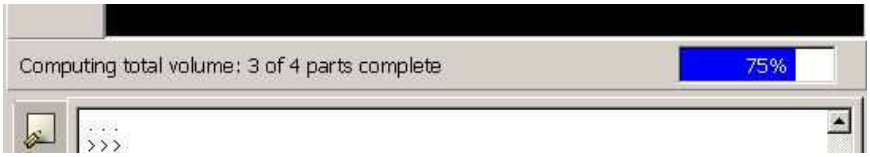


Figure 1: Displaying the progress of a command.

Command error handling

If the command sent to the kernel raises an exception, the mode infrastructure calls the `handleException` method. The `handleException` posts an error dialog with the message contained in the exception. Alternatively, if you want to perform your own error handling, you can redefine the `handleException` method.

You can redefine the `handleException` method as shown in the following example:

```
def handleException(self, exception):

    exceptionType = exception[0]
    exceptionValue = exception[1]

    # Do some special error handling here

    # Post an error dialog
    #

    db = self.getCurrentDialog()
    showAFXErrorDialog(db, str(exceptionValue))
```

Form modes

A form mode gathers input from the user using one or more dialog boxes. This section describes the two methods used by forms for posting dialog boxes.

In this section:

- *Form example*
- *Form constructor*
- *getFirstDialog*
- *getNextDialog*
- *Collecting input from the GUI*

Form example

The following example illustrates how to write a form mode.

This first example contains only one dialog box; a subsequent example will extend this form to include multiple dialog boxes.

```
from abaqusGui import *
from plateDB import PlateDB

class PlateForm(AFXForm):

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self, owner):

        AFXForm.__init__(self, owner)

        self.cmd = AFXGuiCommand(self, 'Plate', 'examples')
        self.nameKw = AFXStringKeyword(self.cmd, 'name', True)
        self.widthKw = AFXFloatKeyword(self.cmd, 'width', True)
        self.heightKw = AFXFloatKeyword(self.cmd, 'height', True)

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def getFirstDialog(self):

        self.cmd.setKeywordValuesToDefaults()
        return PlateDB(self)
```

Form constructor

You begin writing a form mode by deriving a new class from `AFXForm`.

In the body of the `AFXForm` constructor you must call the base class constructor and pass in the owner, which is the module or toolset GUI to which this form belongs.

You then define the commands and keywords that the mode will use. The keywords are stored as members of the mode so that they can be accessed by dialog boxes. If you set `registerQuery`=True in the `AFXGuiCommand` constructor, the mode will query the kernel object specified by the command when it is activated and will automatically set the values of the command's keywords. For more information, see *Keeping the GUI and commands up-to-date*. If there is no kernel object associated with your command (for example, when creating a new object), you can set the keyword values by specifying a default value in the constructor.

If you have a default object that you want to use to reestablish default values for the dialog box, you can use the mode's `registerDefaultsObject` method to register an object whose values will be queried when the user presses the **Defaults** button in the dialog box. For more information, see *Defaults objects*.

By default, dialog boxes are posted as modeless or nonmodal. You can change the behavior by calling `setModal(True)` to have a dialog box posted as modal. In most cases you set the behavior only once; however, you can change the behavior as often as needed by calling the `setModal` method in the `getFirstDialog` or `getNextDialog` methods. For more information, see *Modal versus modeless*.

getFirstDialog

You must write the `getFirstDialog` method for your mode.

The `getFirstDialog` method should return the first dialog box of the mode. In *Form example* a pointer to the form is passed into the dialog box constructor. The dialog box will use this pointer to access the mode's keywords.

If you want the same default values to appear every time you post the dialog box, you must call the `setKeywordValuesToDefaults()` method before returning the dialog box, as shown in *Form example*.

getNextDialog

If your mode contains more than one dialog box, you must write the `getNextDialog` method in addition to the `getFirstDialog` method.

The previous dialog box is passed into the `getNextDialog` method so that you can determine where the user is in the sequence of dialog boxes and act accordingly. The `getNextDialog` method should return the next dialog box in the sequence, or it should return `None` to indicate that it has finished collecting input from the user. The following example is a modified version of the example in *getFirstDialog* that illustrates how inputs are collected from the user in a series of three dialog boxes rather than just one:

```
def getFirstDialog(self):

    self.dialog1 = PlateDB1(self)
    return self.dialog1

def getNextDialog(self, previousDb):

    if previousDb == self.dialog1:
        self.dialog2 = PlateDB2(self)
        return self.dialog2
    elif previousDb == self.dialog2:
        self.dialog3 = PlateDB3(self)
        return self.dialog3
    else:
        return None
```

Collecting input from the GUI

To collect input from the user via the GUI, the keywords defined in the mode must be connected to widgets in the dialog box.

The `AFXDataDialog` class takes a mode argument in its constructor. Because the form stores keywords, the dialog box can access these keywords and assign them to be targets of widgets in the dialog box. As a result, the GUI can update the keywords; or, if the kernel is updated while the dialog box is posted, the keywords can update the GUI. For more information, see *Dialog boxes*. The following example shows how the form's keywords are connected to the widgets in the dialog box:

```
class PlateDB(AFXDataDialog):

    def __init__(self, mode):

        AFXDataDialog.__init__(self, mode, 'Create Plate',
            self.OK|self.CANCEL, DIALOG_ACTIONS_SEPARATOR)

        va = AFXVerticalAligner(self)
        AFXTextField(va, 15, 'Name:', mode.nameKw, 0)
        AFXTextField(va, 15, 'Width:', mode.widthKw, 0)
        AFXTextField(va, 15, 'Height:', mode.heightKw, 0)
```

Procedure modes

A procedure consists of a series of steps that collect input from the user.

In this section:

- *Procedure example*
- *Procedure constructor*
- *getFirstStep*
- *getNextStep*
- *getLoopStep*
- *AFXDialogStep*

Procedure example

This section provides an example of how to write a one-step procedure mode.

Steps in a procedure are posted using the following methods:

- `getFirstStep`
- `getNextStep`
- `getLoopStep`

The following types of steps are available for use in procedures:

- `AFXDialogStep`. This step provides an interface to a dialog box.
- `AFXPickStep`. This step provides an interface to allow picking entities in the viewport.

The following example shows how to write a simple, one-step procedure mode that uses a dialog box step. Subsequent examples will extend this example to show how to use more steps.

```
from abaqusGui import *
from plateDB import PlateDB

class PlateProcedure(AFXProcedure):

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self, owner):

        AFXProcedure.__init__(self, owner)

        self.cmd = AFXGuiCommand(self, 'Plate', 'examples')
        self.nameKw = AFXStringKeyword(self.cmd, 'name', True)
        self.widthKw = AFXFloatKeyword(self.cmd, 'width', True)
        self.heightKw = AFXFloatKeyword(self.cmd, 'height', True)

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def getFirstStep(self):

        self.cmd.setKeywordValuesToDefaults()
        db = PlateDB(self)
        return AFXDialogStep(self, db)
```

Procedure constructor

You begin writing a procedure mode by deriving a new class from `AFXProcedure`.

In the body of the `AFXProcedure` constructor you must call the base class constructor and pass in the owner, which is the module or toolset GUI to which this procedure belongs. Optionally, you can pass in a value for the type of procedure. The default value for the type is NORMAL. The type defines what happens when a new procedure is activated while another procedure is currently executing.

The type of a procedure can be either NORMAL or SUBPROCEDURE. When a normal procedure is activated, it cancels any procedure that is currently executing. When a subprocedure is activated, it suspends a normal procedure or cancels another subprocedure. If a procedure is suspended, it resumes upon the completion of the subprocedure.

View procedures (for example, pan, rotate, and zoom) are special types of procedures that cannot be suspended. View procedures are always cancelled when another procedure is activated, and they always suspend any currently executing procedure when they are activated.

By default, procedures are identified by the infrastructure using their class name. If you need to have multiple instances of a procedure executing at the same time, you will need to distinguish their names to the infrastructure by calling the `setModeName` method.

After you have derived a new class from `AFXProcedure`, you define the commands and keywords needed for the mode. The keywords are stored as members of the mode so that they can be accessed by steps. If you set `registerQuery`=True in the `AFXGuiCommand` constructor, the mode will query the kernel object specified by the command when it is activated and automatically set the values of the command's keywords. For more information, see *Keeping the GUI and commands up-to-date*. If there is no kernel object associated with your command (for example, when creating a new object), you can set the keyword values by specifying a default value in their constructor.

If you have a default object that you want to use to reestablish default values for a dialog box, you can use the mode's `registerDefaultsObject` method to register an object whose values will be queried when the user presses the **Defaults** button in the dialog box. For more information, see *Defaults objects*.

By default, dialog boxes are posted as modeless. You can post a dialog box as modal by calling `self.setModal(True)`. In most cases you set the modality only once in the mode; however, you can change the modality as often as needed by calling the `setModal` method in the `getFirstDialog` or `getNextDialog` methods. For more information, see *Modal versus modeless*.

getFirstStep

You must always write the `getFirstStep` method for your mode. The `getFirstStep` method should return the first step of the mode.

In *Procedure example* a pointer to the procedure is passed into the dialog box constructor. The dialog box will use this pointer to access the mode's keywords.

If you want the same default values to appear every time you post the dialog box, you must call the `setKeywordValuesToDefaults()` method before returning the dialog box, as shown in *Procedure example*.

getNextStep

If your mode contains more than one step, you must write the `getNextStep` method in addition to the `getFirstStep` method.
The previous step is passed into the `getNextStep` method so that you can determine where the user is in the sequence of steps and act accordingly.

The `getNextStep` method should return the next step in the sequence, or it should return `None` to indicate that it has finished collecting input from the user. The following example, which is a modified version of the example in *Procedure example*, illustrates how inputs are collected from the user in a series of three steps rather than just one:

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def getFirstStep(self):

    self.cmd.setKeywordValuesToDefaults()
    self.plateWidthDB = None
    self.plateHeightDB = None
    db = PlateNameDB(self)
    self.step1 = AFXDialogStep(self, db)
    return self.step1

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
def getNextStep(self, previousStep):

    if previousStep == self.step1:
        if not self.plateWidthDB:
            self.plateWidthDB = PlateWidthDB(self)
        self.step2 = AFXDialogStep(self, self.plateWidthDB)
        return self.step2

    elif previousStep == self.step2:
        if not self.plateHeightDB:
            self.plateHeightDB = PlateHeightDB(self)
        self.step3 = AFXDialogStep(self, self.plateHeightDB)
        return self.step3

    else:
        return None
```

getLoopStep

If you want your procedure to loop, you must write the `getLoopStep` method.

The `getLoopStep` method is defined in the base class to return `None`, indicating that the mode will be run through a single time. You can redefine the `getLoopStep` method and return a step to which the procedure should loop back. The following example shows how you can make the procedure shown in the previous section loop back to the first step after it has completed the last step:

```
def getLoopStep(self):
    return self.step1
```

AFXDialogStep

The `AFXDialogStep` class allows you to post a dialog box during a procedure.

To create a dialog step, you must supply the procedure, a dialog box, and, optionally, a prompt for the prompt line. If you do not supply a prompt, Abaqus uses a default prompt of `Fill out the dialog box title dialog`. The following is an example of a dialog step in a single step procedure:

```
def getFirstStep(self):

    db = PlateDB(self)
    prompt = 'Enter plate dimensions in the dialog box'
    return AFXDialogStep(self, db, prompt)
```

In most cases a procedure will have more than one step. Since a procedure has the ability to back up to previous steps, you must write procedures that do not construct dialog boxes more than once during the procedure. You can prevent a procedure from constructing dialog boxes more than once by initializing procedure members and then checking the members during `getNextStep`, as shown in the following example:

```
def getFirstStep(self):

    self.plateWidthDB = None
    self.plateHeightDB = None
    db = PlateNameDB(self)
    self.step1 = AFXDialogStep(self, db)
    return self.step1

def getNextStep(self, previousStep):

    if previousStep == self.step1:
        if not self.plateWidthDB:
            self.plateWidthDB = PlateWidthDB(self)
        self.step2 = AFXDialogStep(self, self.plateWidthDB)
        return self.step2

    elif previousStep == self.step2:
        if not self.plateHeightDB:
            self.plateHeightDB = PlateHeightDB(self)
        self.step3 = AFXDialogStep(self, self.plateHeightDB)
        return self.step3

    else:
        return None
```

Picking in procedure modes

This section describes picking in procedure modes.

In this section:

- *AFXPickStep*
- *Refining what the user can select*
- *Nonpickable entities*
- *Highlighting while selecting*
- *Selection options*
- *Allowing the user to type in points*
- *Picking by angle*
- *AFXOrderedPickStep*
- *Prepopulating a pick step*
- *Limitations while selecting*

AFXPickStep

The `AFXPickStep` class allows the user to pick entities in the current viewport.

You must create the keywords associated with pick steps in the same order as the pick steps in which the keywords are used. For example, if you have two pick steps, you must create the keyword passed into the first pick step before you create the second keyword, which is passed into the second pick step. Creating the keywords associated with pick steps in the same order as the pick steps in which the keywords are used ensures that the necessary setup commands are issued in the proper order for the command to work correctly.

You can specify many parameters when picking items from the viewport. You specify some of these parameters in the `AFXPickStep` constructor, and you specify other parameters by calling various methods of the pick step.

To construct a pick step, you must at least supply the following:

- A procedure
- An object keyword
- A prompt for the prompt line
- A bit flag or flags specifying which type of entities may be picked

The following example shows how you can write a pick step:

```
class MyProcedure(AFXProcedure):

    def __init__(self, owner):

        AFXProcedure.__init__(self, owner)

        self.cmd = AFXGuiCommand(self, 'myMethod', 'myObject')
        self.nodeKw = AFXObjectKeyword(self.cmd, 'node', True)

    def getFirstStep(self):

        return AFXPickStep(self, self.nodeKw,
            'Select a node', AFXPickStep.NODES)
```

Optional parameters in the constructor allow you to specify the following:

- Whether the user should pick one entity or one or more entities (AFXPickStep.ONE, the default, or AFXPickStep.MANY)
- The highlight level (1–4)
- The sequence style (AFXPickStep.ARRAY, the default, or AFXPickStep.TUPLE)

If the user is allowed to pick only one entity, the procedure will automatically advance to the next step after the user picks an entity; however, the user can back up to the previous step to change the selection. If the user is allowed to pick one or more entities, the user must commit the selections by clicking mouse button 2 or by clicking the **Done** button on the prompt line.

The highlight level controls the color of the selected entities. In some procedures, different colors are used between steps to distinguish the selections.

The sequence style controls how a sequence of picked objects is represented in the command string. If the sequence style is AFXPickStep.ARRAY, the picked objects will be represented as the concatenation of slices of arrays; for example, v[3:4] + v[5:8], where v is a vertex array. You cannot use the AFXPickStep.ARRAY sequence style to pick a combination of entities with multiple types because only objects of the same type can be concatenated. In addition, you cannot use the AFXPickStep.ARRAY sequence style to pick interesting points because interesting points are constructed on-the-fly and are not accessible from slices of an array.

If the sequence style is AFXPickStep.TUPLE, the picked objects will be represented as a tuple of individual objects; for example, (v[3], v[5], v[6], v[7]). The style you choose depends on the format accepted by the command that you intend to issue. Some commands in Abaqus/CAE accept both styles, but some accept only one or the other. For further details on the arguments to the AFXPickStep constructor, see the *Abaqus GUI Toolkit Reference Guide*.

Refining what the user can select

A refinement qualifies the types of pickable entities specified in the AFXPickStep constructor.

The following example shows how to select only straight edges:

```
step = AFXPickStep(self, self.edgeKw, 'Select a straight edge',
    AFXPickStep.EDGES)
step.setEdgeRefinements(AFXPickStep.STRAIGHT)
```

By default, no refinements are set. For a complete list of refinements, see the *Abaqus GUI Toolkit Reference Guide*.

Nonpickable entities

By default, the procedure mode prevents previously selected geometric entities from being selected twice in the same procedure. If you do not want this behavior, you can call the `allowRepeatedSelections` method.

The following example shows how to allow repeated selections:

```
step = AFXPickStep(self, self.edgeKw, 'Select a straight edge',
    AFXPickStep.EDGES)
step.allowRepeatedSelections(True)
```

Disallowing repeated picks works only for geometry items such as vertices, edges, and faces; it does not work for nodes and elements.

Highlighting while selecting

The procedure mode clears all highlighting when the user cancels a procedure. In addition, the procedure mode clears highlighting in the current step before backing up.

The color of highlighted entities is controlled by the highlight level set in the AFXPickStep constructor.

Selection options

The **Selection Options** dialog box is automatically available in any pick step.

The available options in the **Selection Options** dialog box are automatically configured, based on the types of entities that the user is picking. For example, if the user is picking only faces, only **Faces** appears in the combo box in the dialog box. Similarly, if the user is picking a single entity, the drag shape and drag scope buttons are not available. As a result, procedures generally do not need to set the available selection options explicitly. If you need to set these options, you can use the procedure's `setSelectionOptions` method. You must set the procedure selection options prior to creating the first pick step. For more information, see the *Abaqus GUI Toolkit Reference Guide*.

Normally a procedure will set these options only at the start of the procedure. However, during the procedure the user may change the settings, and the modified settings will be retained from step to step during the rest of the procedure.

Allowing the user to type in points

If you want to allow the user to type in the coordinates of a point as an alternative to picking in the viewport, you can call the `addPointKeyIn` method and pass it a tuple keyword.

The `addPointKeyIn` method posts a text field on the prompt line. The type of the keyword passed into the `addPointKeyIn` method determines what values are collected from the user; for example, two or three values and whether those values are float or integer types. For example, in the constructor of your procedure you could define an additional keyword as shown in the following code:

```
    self.pointKw1 = AFXObjectKeyword(self.cmd, 'point', True)
    self.pointKw2 = AFXTupleKeyword(self.cmd, 'point', True,
        3, 3, AFXTUPLE_TYPE_FLOAT)
```

In one of the steps of your procedure you could add a key-in option, as shown below:

```
    step = AFXPickStep(self, self.pointKw1, 'Select a point',
        AFXPickStep.POINTS)
    step.addPointKeyIn(self.pointKw2)
```

If a step has a key-in text field, the user enters some values in the text field, and the user commits the values by pressing `Enter`, those values will be used in the command. Alternatively, if a step has a key-in text field and the user selects an entity in the viewport, that entity will be used in the command, regardless of whether anything was typed in the text field. The mode automatically takes care of deactivating whichever keyword needs to be deactivated based on these rules. In the previous example, if the user types in a point, `self.pointKw1` will be deactivated and `self.pointKw2` will be activated. In addition, `self.pointKw2` will contain the value entered by the user.

Picking by angle

Picking by face angle or by edge angle is always enabled when appropriate.

For example, picking by face angle is enabled when the user is picking faces. You cannot disable picking by angle.

AFXOrderedPickStep

The AFXOrderedPickStep is a special pick step that preserves the order in which the user picks entities.

For example, when picking four nodes to create a quad element, the order in which the user picks the nodes is important and must be preserved during picking. The user must pick the entities one at a time and cannot drag select them. In addition, because this is a single step that treats the picked entities as a single pick, the user cannot backup any of the individual picks. The step continues to loop until the user clicks the mouse button two.

Prepopulating a pick step

In some cases you may want to prepopulate a pick step with some selections.

For example, let's say that you have a procedure that creates an object that includes selecting a region of the model and you want to allow the user to edit that object. In the edit procedure you will want to prepopulate the region selection step with the selection that the user originally made when creating the object. You can do this by adding a set of entities to the pick step as shown below:

```
step = AFXPickStep(self, self.nodesKw, 'Select some nodes',
    AFXPickStep.NODES, AFXPickStep.MANY, 1, AFXPickStep.ARRAY)
step.addNodeSetSelection('NodeSet-1')
```

There are similar methods for adding entities from geometry sets (`addGeometrySetSelection`), element sets (`addElementSetSelection`), and surfaces (`addSurfaceSelection`). When this step is executed the added entities will automatically be highlighted, and the user can add or remove from that selection.

Limitations while selecting

Limitations apply to picking procedures.

The following limitations apply:

- The following entities cannot be picked:
  - Sets
  - Surfaces
- Picking more than one kind of entity at the same time is not supported for a sequence style of ARRAY; for example, the user cannot pick nodes and elements in the same step.
- Picking Features or Instances cannot be combined with picking other types of entities. In addition, a sequence style of ARRAY is not supported.
- There is no support for unselecting entities that belong to selected entities. For example, when the user selects a face, Abaqus also selects all the edges belonging to the selected face. The user cannot unselect one of those edges.
- Probing is not supported.

These limitations may be removed in a future release of the Abaqus GUI Toolkit.

GUI modules and toolsets

This part describes how you can create your own modules and toolsets. This part also describes how you can modify an existing Abaqus/CAE module or toolset.

In this section:

- *Creating a GUI module*
- *Creating a GUI toolset*
- *Customizing an existing module or toolset*

Creating a GUI module

This section describes how you can create a GUI module.
To create a new GUI module, you must follow these steps:

- Derive a new class from a module base class.
- Create menus in the menu bar.
- Create icons in the toolbar. This step is optional.
- Create icons in the toolbox. This step is optional.
- Create modes to collect input from the user and issue commands. Modes include procedures and dialog boxes.
- Create methods to handle any special behavior not handled by the module's modes. This step is optional.

These steps are described in detail in the following sections.

In this section:

- *Examining a GUI module example*
- *Registering a GUI module*
- *Switching to a GUI module*

Examining a GUI module example

This section shows how to create a module GUI.

In this section:

- *GUI module example*
- *Deriving a new module class*
- *Tree tabs*
- *Menu bar items*
- *Toolbar items*
- *Toolbox items*
- *Registering toolsets*
- *Kernel module initialization*
- *Instantiating the GUI module*

GUI module example

This section provides an example of how to create a GUI module.

The AFXModuleGui base class provides various module infrastructure support functions. For example, the AFXModuleGui base class keeps track of the module's menus, along with its toolbar and toolbox icons. As a result, the menus, toolbars, and icons can be swapped in and out automatically as the user changes modules.

The following example shows how to create a module GUI; subsequent sections explain the details of this example.

```
from abaqusGui import *
from myModes import mode_1, mode_2, mode_3
from myIcons import *
from myToolsetGui import MyToolsetGui
class MyModuleGui(AFXModuleGui):
    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self):

        # Construct the base class
        #
        mw=getAFXApp().getAFXMainWindow()
        AFXModuleGui.__init__(self, moduleName='My Module',
            displayTypes=AFXModuleGui.PART)
        mw.appendApplicableModuleForTreeTab('Model',
            self.getModuleName() )
        mw.appendVisibleModuleForTreeTab('Model',
            self.getModuleName() )


        # Menu items
        #
        menu = AFXMenuPane(self)
        AFXMenuTitle(self, '&Menu1', None, menu)
        AFXMenuCommand(self, menu, '&Item 1', None, mode_1,
            AFXMode.ID_ACTIVATE)

        subMenu = AFXMenuPane(self)
        AFXMenuCascade(self, menu, '&Submenu', None, subMenu)
        AFXMenuCommand(self, subMenu, '&Subitem 1', None, mode_2,
            AFXMode.ID_ACTIVATE)

        # Toolbar items
        #
        group = AFXToolbarGroup(self)
        icon = FXXpmIcon(getAFXApp(), iconData1)
        AFXToolButton(group, '\tTool Tip', icon, mode_1,
            AFXMode.ID_ACTIVATE)

        # Toolbox items
        #
        group = AFXToolboxGroup(self)
        icon = FXXPMIcon(getAFXApp(), iconData2)
        AFXToolButton(group, '\tTool Tip', icon, mode_1,
            AFXMode.ID_ACTIVATE)

        popup = FXPopup(getAFXApp().getAFXMainWindow())
        AFXFlyoutItem(popup, '\tFlyout Button1', squareIcon,
            mode_1, AFXMode.ID_ACTIVATE)
        AFXFlyoutItem(popup, '\tFlyout Button 2', circleIcon,
            mode_2, AFXMode.ID_ACTIVATE)
        AFXFlyoutItem(popup, '\tFlyout Button 3', triangleIcon,
```

```
            mode_3, AFXMode.ID_ACTIVATE)
        AFXFlyoutButton(group, popup)

        # Register toolsets
        #
        self.registerToolset(MyToolsetGui(),
            GUI_IN_MENUBAR|GUI_IN_TOOL_PANE)

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def getKernelInitializationCommand(self):
        return 'import myModule'

# Instantiate the module
#
MyModuleGui()
```

Deriving a new module class

To create your own module GUI, you begin by deriving a new class from the AFXModuleGui base class. Alternatively, if there is another module GUI class that provides most of the functionality that you want, you can begin by deriving from that class and then make modifications.
For more information, see *Customizing an existing module or toolset*.

Inside the new class constructor body, you must call the base class constructor and pass `self` as the first argument. The `moduleName` is a string used by the GUI infrastructure to identify this module. `displayTypes` are the flag or flags that specify the type of object that is being displayed in this module. Possible values are AFXModuleGui.PART, AFXModuleGui.ASSEMBLY, AFXModuleGui.ODB, AFXModuleGui.XY_PLOT, and AFXModuleGui.SKETCH. If you specify AFXModuleGui.ASSEMBLY, your module must import the assembly kernel module because the assembly kernel module is required to initialize some assembly display options. For more information, see *Kernel module initialization*.

Tree tabs

By default, the tabs in the `TreeToolsetGui` are not visible when the user switches into a custom module. To specify that a tab should be visible or applicable to a module, use the `appendApplicableModuleForTreeTab` and `appendVisibleModuleForTreeTab` methods.

The example in *GUI module example* specifies that the Model tab will be applicable and visible in "My Module." If the user is in the Part module and switches to "My Module," the **Results** tab will be hidden, and the **Model** tab will be made current if it was not already current.

Menu bar items

Menu bar items consist of a menu title that controls a menu pane.
The menu pane, in turn, contains menu commands. Menu commands are buttons that invoke modes.

The example in *GUI module example* creates a top-level menu pane that contains a submenu. The menu commands in the submenu specify the mode that the menu command will invoke by sending the mode an activate message. For more information, see *Mode processing*. *Figure 1* shows the menus and submenus created by the example.

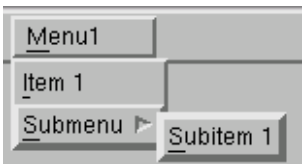


Figure 1: Creating menus.

Toolbar items

Toolbar items are displayed across the top of the main window under the menu bar and consist of a button that contains an icon.
Toolbar items are placed in a group that is shown only when its module or toolset is current. The group also contains a separator that provides a visual distinction from the other groups of icons in the toolbar.

The example in *GUI module example* creates a toolbar group and adds a button to the toolbar. The new button invokes the same mode that will be invoked by the first menu item in the example. For more information, see *Mode processing*.

Toolbox items

Toolbox items are displayed along the left edge of the main window and consist of a button that contains an icon. As with toolbar items, toolbox items are placed in a group that is shown only when its module or toolset is current. Similarly, toolbox groups are spaced apart to provide a visual distinction from the other groups of icons in the toolbox.

The example in *GUI module example* creates a toolbox group and adds a button to the toolbox. The new button invokes the same mode as the first menu item in the example.

Toolboxes can also contain flyout menus. When the user presses mouse button 1 on the flyout button and holds it down for a certain time span, a flyout button displays a popup window containing buttons. If the user just quickly clicks mouse button 1 on the flyout button, the flyout popup is not displayed and the flyout button acts as a regular button. A flyout button displays an icon for the current function along with a triangle in the lower right corner. *Figure 1* shows the flyout buttons created by the example.

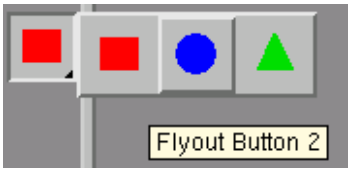


Figure 1: Toolbox flyout buttons.

Registering toolsets

Modules can include toolsets simply by registering them.
Toolsets that are registered with a module will be available when that module is the current module.

To register a toolset, you supply a pointer to the toolset along with bit flags that specify where in the GUI the toolset has defined components. The following locations are supported:

FLAG	LOCATION IN GUI
GUI_IN_NONE	Toolset has no components in standard locations.
GUI_IN_MENUBAR	Toolset has components in the menu bar.
GUI_IN_TOOL_PANE	Toolset has components in the **Tools** menu pull down pane.
GUI_IN_TOOLBAR	Toolset has components in the toolbar.
GUI_IN_TOOLBOX	Toolset has components in the toolbox.

The example in *GUI module example* registers a toolset that contains elements in the main menu bar and the Tools menu.

If you do not specify a flag in an area in which you have created some GUI components, those components will not be shown in the application.

Kernel module initialization

In general, a GUI module is designed to provide an interface to a kernel module.
After the GUI collects input from the user, it constructs a command string that is sent to the kernel for processing. For the command to be recognized on the kernel side, the appropriate kernel module must have been imported before the command was sent.

When a GUI module is loaded for the first time, a special method named `getKernelInitializationCommand` is executed. This method is empty in the base class implementation, and it is up to you to write a method that returns the proper command that will import the appropriate modules on the kernel side. The appropriate modules include any module for which your GUI module can issue commands. If more than one module is required, you can separate the statements by semi-colons or "\n" characters. To avoid name space conflicts with modules loaded by Abaqus, you should use the `import moduleName` style for importing modules and not the `from moduleName import *` style, as shown in the example in *GUI module example*.

Instantiating the GUI module

The final step in the module GUI code is to construct the module.

You can construct the module by calling the module constructor at the end of the module GUI file. This will construct all the objects defined in the constructor body For example,

```
MyModuleGui()
```

Registering a GUI module

To make a GUI module accessible to the GUI infrastructure, you must register the module in the main window code. The register command takes two arguments: one for the name to be displayed in the **Module** combo box in CAE, and a second that specifies the name of the module to import.

For more information, see *Registering modules*. In most cases you register the module in the main window code by calling the module constructor at the end of the module GUI file.

If the example shown in *GUI module example* resides in a file named `myModuleGui.py`, `myModuleGui` must be used as the second argument to the `registerModule` method, as shown in the following statement:

```
# Register modules
#
self.registerModule(
    displayedName='My Module', moduleImportName='myModuleGui')
```

Switching to a GUI module

When the user selects a module from the **Module** list in the context bar, the GUI infrastructure calls the `deactivate` method of the current GUI module and calls the `activate` method of the GUI module selected by the user.

You can write your own `activate` or `deactivate` methods if you need to perform any special processing when entering or leaving a module. If you need to issue a command to the kernel when the user changes modules, you must use the `sendCommandString` method of the AFXModuleGui object to issue the command. If you do not use the `sendCommandString` method, the application may hang while trying to process the command. You should enclose the statements that call the `sendCommandString` method in a `try` block to catch any exceptions generated by the kernel command.

To switch to a GUI module using a script, you can use the `switchModule` method. For example, if you want to switch to your module upon application startup, you can add the following line to the application startup file:

```
switchModule('My Module')
```

This line should appear just before the `app.run()` statement.

You can use the `setSwitchModuleHook(`*function*`)` method to set a callback function that will be invoked when the user switches into a GUI module. Every time the user switches into a GUI module, your function will be called and the name of the module will be passed into the function. For example,

```
def onModuleSwitch(moduleName):
    if moduleName == 'Part':
       # Do part module tasks
    elif moduleName == 'Mesh':
       # Do mesh module tasks
    etc.

setSwitchModuleHook(onModuleSwitch)
```

Creating a GUI toolset

Toolsets are similar to modules, except that they can be used in more than one module. Toolsets typically have less functionality than modules because toolsets specialize in performing a specific task, such as partitioning. To create a new GUI toolset, you must follow these steps:

- Derive a new class from a toolset base class.
- Create menus in the menu bar. This step is optional.
- Create items in the Tools menu. This step is optional.
- Create buttons in the toolbar. This step is optional.
- Create buttons in the toolbox. This step is optional.
- Create modes to collect input from the user and to issue commands.

In this section:

- *GUI Toolset example*
- *Creating toolset components*
- *Registering toolsets*

GUI Toolset example

The AFXToolsetGui base class provides various toolset infrastructure support functions.

For example, the AFXToolsetGui base class keeps track of the menus in a toolset, along with the toolbar and toolbox buttons, so that they can be swapped in and out automatically as the user changes modules. To create your own toolset GUI, you begin by deriving from the AFXToolsetGui class. Alternatively, if there is another module GUI class that provides most of the functionality that you want, you can begin by deriving from that class and then making modifications. For more information, see *Customizing an existing module or toolset*.

The following example shows how to create a new toolset class by deriving from AFXToolsetGui:

```
from abaqusGui import *

class MyToolsetGui(AFXToolsetGui):

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self):

        # Construct the base class
        #
        AFXToolsetGui.__init__(self, toolsetName)
        ...
```

In the constructor of the new class you call the constructor of the base class. The AFXToolsetGui class takes the following argument:

toolsetName

A String specifying the name of the toolset. The toolset name provides an identifier for the toolset.

Creating toolset components

You create menu, toolbar, and toolbox items in a toolset in the same way that you create those items in a module.

When you create menu, toolbar, and toolbox items in a module, the module is used as the parent. In contrast, when you create a toolset component such as menu panes, the toolset is used as the parent of the toolset component. The toolset is used as the parent because the components need to be managed by the toolset when the toolset is swapped in and out of the GUI. For more information on creating these components, see *Creating a GUI module*.

Registering toolsets

You can register a toolset with the main window or with a module.

If you register the toolset with the main window, the toolset will be persistent throughout the session; for example, the File toolset is always available in an Abaqus/CAE session. In contrast, if you register the toolset with a module, the toolset will be swapped in and out with that module's menus and icons; for example, the Datum toolset is available in an Abaqus/CAE session only in selected modules. For more information, see *Registering toolsets*.

Customizing an existing module or toolset

The previous sections describe how you can create a new module or toolset by starting from an empty base class and adding all the functionality that you need. Alternatively, you may find that you want to use most of the functionality of an existing module or toolset. If a suitable module or toolset exists, it may be easier for you to derive a new module or toolset from it and then add or remove functionality from it. This chapter describes how to make various modifications to an existing module or toolset.

In this section:

- *Modifying and accessing Abaqus/CAE GUI modules and toolsets*
- *The File toolset*
- *The Tree toolset*
- *The Selection toolset*
- *The Help toolset*
- *An example of customizing a toolset*

Modifying and accessing Abaqus/CAE GUI modules and toolsets

Deriving a new class to create modified Abaqus/CAE modules and toolsets allows you to customize existing functions without changing the original functions. You can also access existing Abaqus/CAE functions from within new dialogs that you create with the Abaqus GUI Toolkit.

In this section:

- *Abaqus/CAE GUI modules and toolsets*
- *Accessing Abaqus/CAE functions*

Abaqus/CAE GUI modules and toolsets

The Abaqus GUI Toolkit is designed to allow you to add your own modules and toolsets.
It is generally not recommended that you modify Abaqus/CAE modules and toolsets because future changes to Abaqus/CAE may “break” your application. However, if you do have a need to modify some of the Abaqus/CAE modules or toolsets, you can make changes by deriving a new class from one of them and then adding or removing components.

To derive a new class, you must know the appropriate class name and you must call that class's constructor in your constructor. The table below lists the class names and registered names for all the Abaqus/CAE modules that are available in the Abaqus GUI Toolkit. You can import these class names from `abaqusGui`.

When you register a module derived from one of the Abaqus/CAE modules, you must use the name shown in the table for the *displayedName* argument in the main window's `registerModule` method. If you do not use the name shown, some GUI infrastructure components may not function correctly.

Class name	Name
PartGui	“Part”
PropertyGui	“Property”
AssemblyGui	“Assembly”
StepGui	“Step”
InteractionGui	“Interaction”
LoadGui	“Load”
MeshGui	“Mesh”
OptimizationGui	“Optimization”
JobGui	“Job”
VisualizationGui	“Visualization”
SketchGui	”Sketch”

When you register a toolset, you must specify in the `registerToolset` method in which locations (the menu bar, the toolbar, or the toolbox) the toolset creates the widget. If you omit a toolset location flag, the GUI for that toolset will not appear in that location. The table below shows the class name for each of the Abaqus/CAE toolsets along with the flags that indicate the locations in which the toolset creates the widgets. You can import these class names from `abaqusGui`.

To register the plug-in toolset, you call `registerPluginToolset()`; you do not use the `registerToolset` method.

When you unregister a toolset, you must use the name shown in the table as the argument to the module's `unregisterToolset` method.

Class name	Name	Toolset locations
AmplitudeToolsetGui	“Amplitude”	GUI_IN_TOOL_PANE
AnnotationToolsetGui	“Annotation”	GUI_IN_MENUBAR | GUI_IN_TOOLBAR
CanvasToolsetGui	“Canvas”	GUI_IN_MENUBAR
CustomizeToolsetGui	“Customize”	GUI_IN_TOOL_PANE
DatumToolsetGui	“Datum”	GUI_IN_TOOLBOX | GUI_IN_TOOL_PANE
EditMeshToolsetGui	“Mesh Editor”	GUI_IN_TOOLBOX | GUI_IN_TOOL_PANE
FileToolsetGui	“File”	GUI_IN_MENUBAR | GUI_IN_TOOLBAR

Class name	Name	Toolset locations
HelpToolsetGui	“Help”	GUI_IN_MENUBAR | GUI_IN_TOOLBAR
ModelToolsetGui	“Model”	GUI_IN_MENUBAR
PartitionToolsetGui	“Partition”	GUI_IN_TOOLBOX | GUI_IN_TOOL_PANE
QueryToolsetGui	“Query”	GUI_IN_TOOLBAR | GUI_IN_TOOL_PANE
RegionToolsetGui	“Region”	GUI_IN_TOOL_PANE
RepairToolsetGui	“Repair”	GUI_IN_TOOLBOX | GUI_IN_TOOL_PANE
SelectionToolsetGui	“Selection”	GUI_IN_TOOLBAR
TreeToolsetGui	“Tree”	GUI_IN_MENUBAR
ViewManipToolsetGui	“View Manipulation”	GUI_IN_MENUBAR | GUI_IN_TOOLBAR

For an example of how to register the toolsets and modules used by Abaqus/CAE, see *Main window example*. The following statements show how you could add your own toolset to the Visualization module:

```
# File myVisModuleGui.py:

    from abaqusGui import *
    from myToolsetGui import MyToolsetGui

    class MyVisModuleGui(VisualizationGui):

        def __init__(self):

            # Construct the base class.
            #
            VisualizationGui.__init__(self)

            # Register my toolset.
            #
            self.registerToolset(MyToolsetGui(),
                GUI_IN_MENUBAR|GUI_IN_TOOLBOX)

    MyVisModuleGui()

# File myMainWindow.py:

    from abaqusGui import *

    class MyMainWindow(AFXMainWindow):

        def __init__(self, app, windowTitle=''):

            ...
            self.registerModule('Visualization',
                'myVisModuleGui')
            ...
```

If you derive a toolset from an Abaqus/CAE toolset, you must construct that toolset using the `makeCustomToolsets` method of `AFXMainWindow`. You must use the `makeCustomToolsets` method to ensure that the toolset is created at the appropriate time during application startup. This will avoid any conflicts with Abaqus/CAE modules that also make use of the module. For example, if you derive a new toolset from the Datum toolset, you must create the new toolset in `makeCustomToolsets`. This approach is illustrated in the following example. The new toolset will also appear in the Part module in place of the standard Datum toolset.

```
# In your main window file:
```

```
    class MyMainWindow(AFXMainWindow):

        def __init__(self, app, windowTitle=''):

              ...
        def makeCustomToolsets(self):

            from myDtmToolsetGui import MyDtmGui
            # Store the toolset as a member of the main window if
            # you want to register it in one of your modules too.
            #
            self.myDtmGui = MyDtmGui()

# In your module GUI file:

      class MyModuleGui(AFXModuleGui):

          def __init__(self):

              ...
              mw = getAFXApp().getAFXMainWindow()
              self.registerToolset(mw.myDtmGui,
                 GUI_IN_TOOL_PANE|GUI_IN_TOOLBOX)
```

Accessing Abaqus/CAE functions

If you want to launch an Abaqus/CAE function from your own dialog, you can do so by connecting the appropriate target and selector to one of your buttons.

You can get the target and selector for a particular function by using the main window's `getTargetFromFunction` and `getSelectorFromFunction` methods. For example:

```
    mainWindow = getAFXApp().getAFXMainWindow()
    target = mainWindow.getTargetFromFunction('Part->Create')
    selector = mainWindow.getSelectorFromFunction('Part->Create')
    FXButton(self, 'Create Part...', tgt=target, sel=selector )
```

The list of valid function names can be found in the **Functions** tab page in the **Tools->Customize** dialog box.

The File toolset

The File toolset contains a method called `getPrintForm` that allows you to access the form that posts the **Print** dialog box.
See *Print dialog box* for an example of how to use the `getPrintForm` method.

In addition, the Abaqus GUI Toolkit provides two virtual methods that you can modify to change the behavior of your application when a database is opened. Normally, after an output database is opened, Abaqus/CAE will enter the Visualization module. Similarly, if you are in the Visualization module and you open a model database, Abaqus/CAE enters the first module listed in the **Module** list in context bar. To change this behavior, you can overwrite the `switchToOdbModule` and `switchToMdbModule` methods. These methods return True if they are successful. For example:

```
from abaqusGui import *

class MyFileToolsetGui(FileToolsetGui):

     def switchToMdbModule(self):

         # Always switch to the Property module
         currentModuleGui = getCurrentModuleGui()
         if currentModuleGui and \
            currentModuleGui.getModuleName() != 'Property':
                 switchModule('Property')
         return True

     def switchToOdbModule(self):

          # Do not switch modules
          return True
```

The Tree toolset

The Tree toolset provides a tabbed area that contains the Model Tree and the Results Tree in Abaqus/CAE. For more information, see *Working with the Model Tree and the Results Tree*. The Tree toolset contains the following methods that you can use to customize the appearance of the tabbed area:

- The `makeModelTab` method creates the tab that contains the Model Tree. The name of the tab is "Model."
- The `makeMaterialLibraryTab` method creates the tab that contains the Material Library. The name of the tab is "Material Library."
- The `makeResultsTab` method creates the tab that contains the Results Tree. The name of the tab is "Results."
- The `makeTabs` method calls all of the methods listed above.

In addition, the main window has an `appendTreeTab` method that creates a new tab item in the tabbed area and returns a vertical frame into which you can add your widgets. If you want to simply add a tab after the Model and Results tabs, you can use `appendTreeTab` from within your custom code. However, if you want to change the order of the tabs or remove one of the standard tabs, you must derive your own toolset from the Tree toolset. For example:

```
class MyTreeToolsetGui(TreeToolsetGui):

    def makeTabs(self):

        self.makeModelTab()
        self.makeMyTab()
        self.makeMaterialLibraryTab()
        self.makeResultsTab()

    def makeMyTab(self):

        vf = getAFXApp().getAFXMainWindow().appendTreeTab(
            'My Tab', 'My Tab')
        FXLabel(vf, 'This is my tab item')
```

The first argument to the `appendTreeTab` method is the text that you want to show up in the tab button. The second argument is the name of the tab, which is used for identification purposes in various application programming interfaces, such as `setCurrentTreeTab`(*name*).

By default, when you create a tab it will be visible in all modules, and it will be applicable to all modules. If you do not want your tab to be visible or applicable to all modules, you can use the `setApplicabilityForTreeTab` and `setVisibilityForTreeTab` methods. When the user switches to a new module, the application will check to see if the current tab is visible in and applicable to the new module. If the tab is not visible, it will be hidden. If it is not applicable, the application will search for the first tab that is applicable to the new module and make that tab current. For example:

```
    def makeMyTab(self):

        vf = getAFXApp().getAFXMainWindow().appendTreeTab(
            'My Tab', 'My Tab')
        getAFXApp().getAFXMainWindow().setApplicabilityForTreeTab(
            'My Tab', 'Part, Property')
        getAFXApp().getAFXMainWindow().setVisibilityForTreeTab(
            'My Tab', 'Part, Property')

        FXLabel(vf, 'This is my tab')
```

In this case, when the user is in the Part module, **My Tab** will be shown. If the user clicks on **My Tab** to make it current and then switches to the Property module, **My Tab** will remain visible and current. If the user switches to the Step module, **My Tab** will be hidden and the **Model** tab will become current (because it has been defined as applicable to all modules except the Visualization module).

The Selection toolset

The Selection toolset provides a selection capability outside of any procedure.
In other words, it allows users to select objects and then invoke a procedure, instead of invoking the procedure and then selecting the objects.

Each module defines a set of entities that can be selected in that module. If you create your own module, then you should set the appropriate selectable entities when your module gets activated. You can use the `getToolset` method of the main window to get the selection toolset, and then use the `setFilterTypes` method of the selection toolset.

- `setFilterTypes(types, defaultType)`

Use the following flags for the `types` and `defaultType` arguments:

- SELECTION_FILTER_NONE
- SELECTION_FILTER_ALL
- SELECTION_FILTER_VERTEX
- SELECTION_FILTER_EDGE
- SELECTION_FILTER_FACE
- SELECTION_FILTER_CELL
- SELECTION_FILTER_DATUM
- SELECTION_FILTER_REF_POINT
- SELECTION_FILTER_NODE
- SELECTION_FILTER_ELEMENT
- SELECTION_FILTER_FEATURE

For example:

```
class MyModuleGui(AFXModuleGui):

    ...

    def activate(self):

        toolset = getAFXApp().getAFXMainWindow().getToolset(
            'Selection')
        toolset.setFilterTypes(
            SELECTION_FILTER_CELL|SELECTION_FILTER_FACE,
            SELECTION_FILTER_FACE)
        AFXModuleGui.activate(self)
```

The Help toolset

The Help toolset contains special methods that allows you to add your own logo and copyright information to the **Help->About Abaqus** dialog box.

Customized applications must show the standard copyright information displayed by Abaqus/CAE or Abaqus/Viewer. In addition, you can customize the copyright information at the top of the **About Abaqus** dialog box using the following methods:

- `setCustomCopyrightStrings(customCopyrightVersion, customCopyrightInfo)`
- `setCustomLogoIcon(logoIcon)`

For example:

```
from abaqusGui import *
from sessionGui import HelpToolsetGui
from myIcons import *
...

class MyMainWindow(AFXMainWindow):

    def _init_(self, app, windowTitle='')

       ...

       # Add custom copyright info to the About Abaqus dialog.
       #
       helpToolset = HelpToolsetGui()
       product = getAFXApp().getProductName()
       major, minor, update = getAFXApp().getVersionNumbers()
       prerelease = getAFXApp().getPrerelease()
       if prerelease:
           release = '%s %s.%s-PRE%s' % (
                product, major, minor, update)
       else:
           release = '%s %s.%s-%s' % (
                product, major, minor, update)
       info = 'Copyright 2003\nMy Company'
       helpToolset.setCustomCopyrightStrings(release, info)
       icon = FXXPMIcon(app, myIconData)
       helpToolset.setCustomLogoIcon(icon)
       self.registerHelpToolset(helpToolset, GUI_IN_MENUBAR)
```

An alternative way to provide help in your application is to use special methods that allow you to post a URL in a web browser. For example:

```
from uti import webBrowser
status = webBrowser.displayURL('http://www.3ds.com/simulia')
status = webBrowser.openWithURL(
    'file://D:/users/someUser/someFile.html')
```

You can use any valid URL syntax, such as “http” or “file.” displayURL will display the URL in a currently open browser window (if there are none, it will open a new window). openWithURL will always open a new browser window. No exceptions are thrown, but you can check the return status of these methods for success.

An example of customizing a toolset

To modify an existing toolset, you start by deriving a new class from it.

To modify widgets in the toolset, you need to be able to access them. The following functions in the Abaqus GUI Toolkit allow you to access a widget:

- `getWidgetFromText(widget, text)`: The `getWidgetFromText` function returns a widget whose label or tip text matches the specified text and is also a child of the specified widget. For example, the following statement returns the widget that matches the **Save As...** item in the **File** menu:

  ```
  saveAsWidget = getWidgetFromText(fileMenu, 'Save As...')
  ```

- `getSeparator(widget, index)`: The `getSeparator` function returns the n^{th} separator of the specified widget, where n is specified by the one-based index. For example, the following statement returns the second separator in the **File** menu:

  ```
  separatorWidget = getSeparator(fileMenu, 2)
  ```

The following example shows how you can modify the **File** toolset GUI. *Figure 1* shows the **File** menu before and after the script is executed.

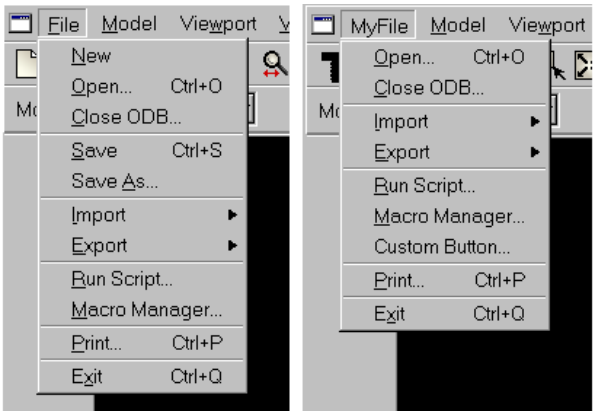


Figure 1: The toolbar and the File menu before and after executing the example script.

```
from sessionGui import FileToolsetGui
from myIcons import boltToolboxIconData
from myForm import MyForm

class MyFileToolsetGui(FileToolsetGui):

    #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    def __init__(self):

        # Construct the base class.
        #
        FileToolsetGui.__init__(self)

        # Remove unwanted items from the File menu,
        # including the second separator.
        #
        menubar = getAFXApp().getAFXMainWindow().getMenubar()
        menu = getWidgetFromText(menubar, 'File').getMenu()
        getWidgetFromText(menu, 'New').hide()
        getWidgetFromText(menu, 'Save').hide()
        getWidgetFromText(menu, 'Save As...').hide()
        getSeparator(menu, 2).hide()
```

```
        # Remove unwanted items from the toolbar
        #
        toolbar = self.getToolbarGroup('File')
        getWidgetFromText(toolbar, 'New Model\nDatabase').hide()
        getWidgetFromText(toolbar, 'Save Model\nDatabase').hide()

        # Add an item to the File menu just above Exit
        #
        btn = AFXMenuCommand(self, menu, 'Custom Button...',
            None, MyForm(self), AFXMode.ID_ACTIVATE)
        sep = getSeparator(menu, 6)
        btn.linkBefore(sep)

        # Rename the File menu
        #
        fileMenu = getWidgetFromText(menubar, 'File')
        fileMenu.setText('MyFile')

        # Change a toolbar button icon
        #
        btn = getWidgetFromText(toolbar, 'Open')
        icon = FXXPMIcon(getAFXApp(), boltToolboxIconData)
        btn.setIcon(icon)
```

This example script illustrates the following:

Deriving a new toolset class

To modify a toolset GUI, you begin by deriving a new class from it. Inside the new class constructor body, you must call the base class constructor and pass *self* as the first argument.

Removing items from a menu or toolbar

You can remove items from a menu by hiding them. You use the `getWidgetFromText` or the `getSeparator` functions to obtain the widgets and call the `hide` method to remove them.

Adding items to a menu

You can insert items into an existing menu by creating new menu commands and positioning them using the `linkBefore` or `linkAfter` methods.

Renaming items and changing icons

You can change the text or icon associated with a widget by calling the `setText` or `setIcon` methods.

Creating a customized application

This part describes how you create a customized application.

In this section:

- *Creating an application*
- *The application object*
- *The main window*
- *Customizing the main window*

Creating an application

This chapter explains how to create an application.

In this section:

- *About the GUI code*
- *Startup script*
- *Licensing and execution*

About the GUI code

An application consists of the two fundamental pieces: the kernel code and the GUI code.

The kernel code consists of Python modules that contain functions and classes for performing various tasks; for example, creating parts or postprocessing results. The GUI code provides a convenient, user-friendly mechanism for gathering the inputs required for the kernel code. Kernel coding is described in the *Abaqus Scripting User's Guide* and the *Abaqus Scripting Reference Guide*.

To develop the GUI code, you begin with a startup script that launches the application from the command line. The script creates an application object, which interacts with the window manager and controls a main window. The main window provides components such as a menu bar, a toolbar, and a toolbox. From that core you add functionality to the application by registering modules and toolsets.

Modules and toolsets are a way of grouping functionality to be presented to the user. For example, the Part module in Abaqus/CAE groups all the functions related to creating and modifying parts. Abaqus/CAE modules and toolsets can be included in your application, and you can write your own modules and toolsets to provide custom functionality.

The widget library provides access to various GUI controls (such as push buttons, check buttons, and text fields) that are used to build dialog boxes. These concepts are illustrated in *Figure 1*. Each of these steps is described in detail in subsequent sections.

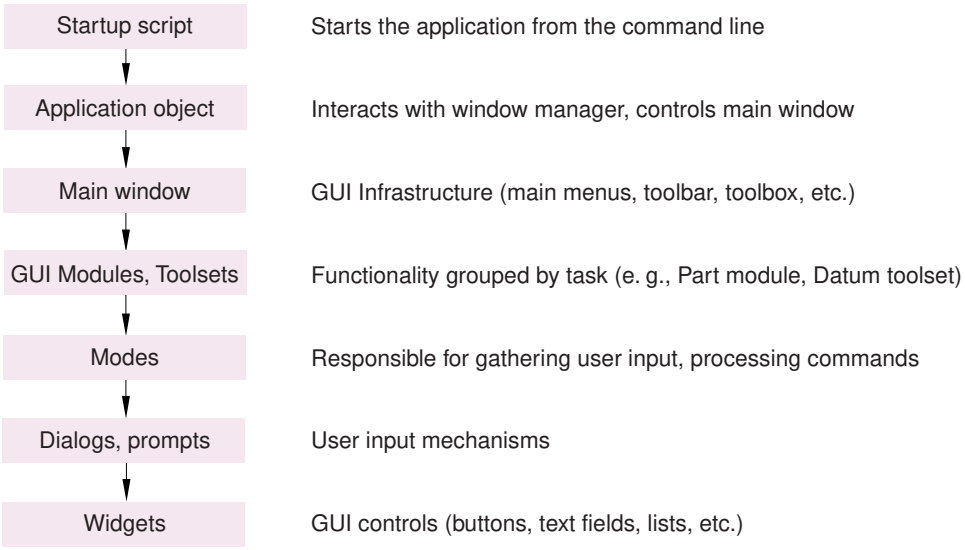


Figure 1: An overview of GUI code.

Startup script

Every application is started from a short startup script.

The startup script performs the following tasks:

- Initializes an application object. The application object is responsible for high-level functions, such as managing message queues and timers and updating the GUI, and controlling the main window. It is not a visible object.
- Instantiates a main window. The main window is what the user sees when the application is first started and provides access to all of the application's functionality.
- Creates and runs the application. Once the application is run, it enters an event loop where it waits to react to user input, such as the click of a mouse.

The following illustrates a typical startup script.

```
from abaqusGui import *
import sys
from caeMainWindow import CaeMainWindow

#Define a custom callback, myStartupCB(), that will be invoked
#once the application has finished its startup processing
#
def myStartupCB():
    from myStartupDB import MyStartupDB
    db = MyStartupDB()
    db.create()
    db.show()

# Initialize the application object
#
app = AFXApp('Abaqus/CAE', 'SIMULIA')
app.init(sys.argv)
# Construct the main window
#
CaeMainWindow(app)

# Create the application
#
app.create()

# Register the custom startup callback
#
# NOTE: This call must be made after app.create()
setStartupCB( myStartupCB )

#Run the application
#
app.run()
```

The first statement in the script imports all constructors, including the `AFXApp` constructor, from the `abaqusGui` module, which contains access to the entire Abaqus GUI Toolkit. The `sys` module is also imported since it is needed to pass arguments into the application's `init` method. After the script imports the `sys` module, it imports the main window constructor.

This startup script has been customized to include a startup callback function that will display a custom dialog, **MyStartupDB**, after the application starts. The callback is defined after the import statements and before the application is initialized. The next statements instantiate and initialize the application object. The application object is discussed in more detail in *The application object*. The script then instantiates the main window. The main window is what the user will see when the application is started. The main window is discussed in more detail in *The main window*.

The application constructor creates all the data structures needed for that object, and the `app.create()` statement creates all the GUI windows required by the application object. Next, the custom callback startup function is registered.

The `app.run()` statement displays the application, including the custom startup dialog, and then enters an event loop. The event loop then waits for user interaction.

When you start your custom application, you may want to use the `-noStartup` option in the Abaqus/CAE execution procedure to prevent Abaqus/CAE from posting its own startup dialog. For more information, see *Abaqus/CAE Execution*.

Licensing and execution

The startup script described in the previous section is run by specifying the name of the script as the argument to the `-custom` option on the command line.

To start your application, enter one of the following,

```
abaqus cae -custom startupScript
abaqus viewer -custom startupScript
```

where *startupScript* is the name of the startup script for your application and does not include a file extension. You are responsible for locating the script in:

- A directory specified in the `PYTHONPATH` environment variable, or
- One of the locations for external plugins, as described in *Where are plug-in files stored?*

The first argument to the `abaqus` command specifies the type of license to be checked out. Specifying `abaqus cae` will check out a token named `cae` that will give you access to all the Abaqus/CAE kernel modules. Specifiying `abaqus viewer` will check out a token called `viewer` that will give you access to only the visualization kernel module. Therefore, if your application needs to import any Abaqus module other than the Visualization module, you must check out a `cae` token.

The application object

This section describes the Abaqus application object. The application object manages the message queue, timers, chores, GUI updating, and other system facilities.

In this section:

- *The application object*
- *Common methods*

The application object

The application object manages the message queue, timers, chores, GUI updating, and other system facilities.

Each application will have an application object, which you typically create in the application's startup file. For more information, see *Startup script*. The constructor for the application object takes the following arguments:

```
AFXApp(appName, vendorName, productName, majorNumber, minorNumber,
       updateNumber, prerelease)
```

The application and vendor names are intended to be keys into the registry. The registry is a place to store settings that are persistent between sessions of the application; for example, the size and location of the application on the desktop when the application it starts. The registry is currently not used by Abaqus, but these keys are included as placeholders for future capabilities. The registry will have various sections that allow you to group settings. Some settings may apply to all products from a particular vendor, and some settings may apply to only a specific product from a vendor.

By default, Abaqus displays the product name and release numbers in the main window's title bar; for more information, see *The title bar*.

Common methods

You can access the application object using the following statement: `app = getAFXApp()`.

The following list shows some of the most commonly used application methods:

getAFXMainWindow()

Returns a handle to the main window object.

getProductName()

Returns the product name.

getVersionNumbers()

Returns a tuple of (*majorNumber*, *minorNumber*, *updateNumber*).

getPrerelease()

Returns True if this application is a prerelease.

beep()

Rings the system bell.

The main window

This section describes the layout, components, and behavior of the Abaqus main window.

In this section:

- *About the main window*
- *The title bar*
- *The menu bar*
- *Toolbars*
- *The context bar*
- *The module toolbox*
- *The drawing area and canvas*
- *The prompt area*
- *The message area*
- *The command line interface*

About the main window

Interactive Abaqus products consist of a single main window that contains several GUI infrastructure components. The main window itself provides only GUI infrastructure support. You add specific functionality to the application by registering modules and toolsets with the main window. Registering modules and toolsets is discussed in detail in *Modules and toolsets*.

The main window is designed to work with the concept of GUI modules, which contain their own menu bar, toolbar, and toolbox entries. The main window shows only the components for one module at a time. The main window is responsible for swapping these components in and out as the user visits the various modules of the application.

The following statement shows the constructor that you use to create the main window:

```
AFXMainWindow(app, title, icon=None, miniIcon=None,
    opts=DECOR_ALL, x=0, y=0, w=0, h=0)
```

The following list describes the arguments to the AFXMainWindow constructor:

app

The application object.

title

A String that will be shown in the title bar of the main window.

icon

A 32 × 32 pixel icon used for the application on the desktop.

miniIcon

A 16 × 16 pixel used on Windows for the application in the title bar and system tray.

opts

Flags controlling various window behavior.

x,y,w,h

The *X*-, *Y*-location of the window, and the *width* and *height* of the window. The default value of zero indicates that the system should calculate these numbers automatically. The main window size and location are stored in `abaqus_2026.gpr` when the application exits so that when the application is started again it will appear in the same location with the same size. Therefore, it is recommended that you do not set *x, y, w, or h* in the main window constructor; however, if you do, those settings will override the settings in `abaqus_2026.gpr`.

The following statement shows how you can access the main window:

```
mainWindow = getAFXApp().getAFXMainWindow()
```

The layout of the main window is shown in *Figure 1*.

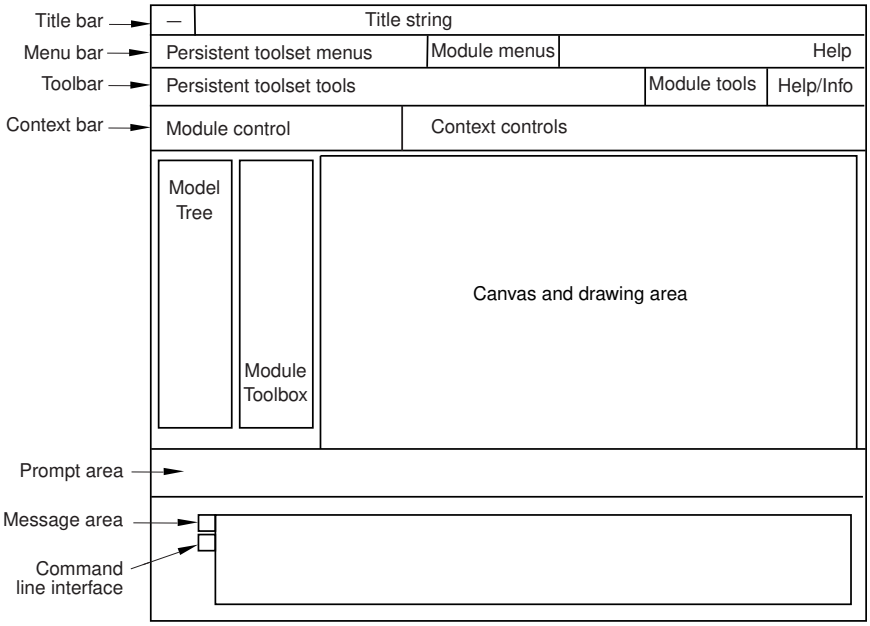


Figure 1: The main window.

The title bar

By default, the string shown in the title bar is constructed from the arguments passed into the AFXApp constructor.

For example,

```
AFXApp(appName, vendorName, productName,
    majorNumber, minorNumber, updateNumber, prerelease)
```

where `majorNumber` is the version number, `minorNumber` is the release number, `updateNumber` is the update number, and `prerelease` is the prerelease number.

The title is generated using the format shown in the following statement:

```
productName + majorNumber + '.' + minorNumber + \
    '-' + updateNumber
```

For example, the following statement,

```
AFXApp(productName="Abaqus/CAE",
    majorNumber=6, minorNumber=12, updateNumber=1)
```

generates the following string in the title bar: `Abaqus/CAE 6.12-1`

If you do not specify the major, minor, and update numbers in the application constructor, they default to the current Abaqus/CAE release numbers. Similarly, if you specify release numbers but you do not specify a product name, the release numbers default to the current Abaqus/CAE release numbers. If you set the `prerelease` argument in the AFXApp constructor to True, the update number is preceded by `PRE`. For example, `Abaqus/CAE 6.12-PRE1`.

In addition, if the user has opened a model database, the title bar string contains the name of the current model database; for example, `Abaqus/CAE 6.12-1 MDB: C:\projects\cars\engines\turbo-1.cae`.

If the name of the current model database, including the path, exceeds 50 characters, the name will be abbreviated by showing only the first and last 25 characters separated by "…".

If you do not want the default title processing, you can override it by specifying a title in the AFXMainWindow constructor. If you specify a title in the AFXMainWindow constructor, the Abaqus GUI Toolkit ignores the arguments in the application constructor and uses the title specified. The model database name and the name of the current viewport (when maximized) will continue to be shown in the title bar, even if you override the default title processing.

You can access the string shown in the window title bar using the following statement:

```
title = getAFXApp().getAFXMainWindow().getTitle()
```

The menu bar

The menu bar consists of the following three areas: persistent toolset menus, module menus, and the help menu.

The persistent toolset menus and the help menu are shown when the application is first started, and they remain visible throughout the user's session. The module menus reflect the current module and are swapped in and out as the user visits the various modules. You can access the menu bar using the following statement:

```
menubar = getAFXApp().getAFXMainWindow().getMenubar()
```

Toolbars

By default, Abaqus/CAE displays all of the toolbars in a row underneath the main menu bar.

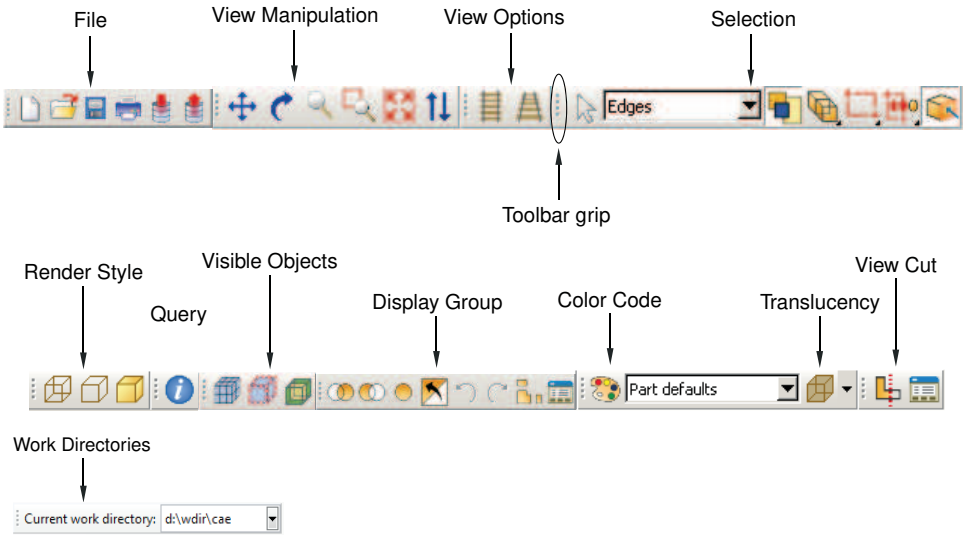


Figure 1: The Abaqus/CAE toolbars.

You can use the following statement to access a toolbar group from the module or toolset that defines the toolbar group:

```
toolbar = self.getToolbarGroup(toolbarName)
```

where *self* is the module or toolset, and *toolbarName* is the name given to the toolbar when Abaqus/CAE constructs it. You can determine the names of the toolbars by selecting **Tools->Customize** from the main menu bar and viewing the dialog box that appears.

The context bar

The context bar contains controls for the current module and other context; for example, the current part.

You can access the context bar using the following statement:

```
contextBar = getAFXApp().getAFXMainWindow().getContextBar()
```

The module toolbox

The module toolbox contains icons for tools commonly used in the current module.

When you switch into a module, that module's toolbox icons replace those of the previous module. You can access the module toolbox using the following statement:

```
toolbox = getAFXApp().getAFXMainWindow().getToolbox()
```

The drawing area and canvas

The canvas provides an infinite space upon which you can create and manipulate viewports. The drawing area is a window into the visible part of the canvas.

You can access the canvas area using the following statement:

```
canvas = getAFXApp().getAFXMainWindow().i_getCanvas()
```

The “i_” in the method name indicates that this is an internal method that you should not normally use—it is expected that only the GUI infrastructure needs to access this method.

The prompt area

The prompt area displays prompts to guide the user, as well as work-in-progress (WIP) messages.

You can display messages in the prompt area during procedures. For more information, see *Picking in procedure modes*.

The message area

The application uses the message area to display informational and warning messages.

You can send messages to the message area using the following method:

```
mainWindow = getAFXApp().getAFXMainWindow()
mainWindow.writeToMessageArea('Warning: Some items failed!')
```

The command line interface

The command line interface (CLI) provides an interface to the kernel-side Python command interpreter.

The CLI does not provide any access to the GUI-side Python interpreter. The user can enter Abaqus scripting interface commands in the CLI, which are then sent to the kernel for processing. In addition, the user can enter standard Python commands in the CLI. For example, the user can use the CLI as a simple calculator, as shown in *Figure 1*.

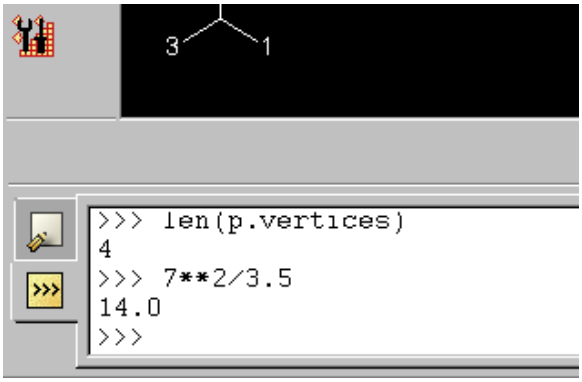


Figure 1: The command line interface.

The Abaqus GUI Toolkit does not expect users to use the CLI to issue Abaqus Scripting Interface commands. Normally all commands sent from the GUI process are sent by the GUI via modes. For more information, see *Modes*. You can hide the CLI if it is not used by your application, as shown in the following statements:

```
mainWindow = getAFXApp().getAFXMainWindow()
mainWindow.hideCli()
```

Customizing the main window

The main window base class provides the basic infrastructure for the application's user interface. For example, you can use this base class to enable user interaction in the interface, manipulate modules and toolsets, and display objects in the viewport. However, this base class does not provide access to any functionality in the application; for example, you cannot use the base class to create parts or run an analysis job.

This section describes how you can assign functionality to the application by deriving from the main window base class and then registering modules and toolsets.

In this section:

- *Modules and toolsets*
- *The Abaqus/CAE main window*

Modules and toolsets

Modules are one of the fundamental concepts of an interactive Abaqus application. A module serves to group functionality into logical units; for example, a unit that creates parts or a unit that meshes the assembly.

An interactive Abaqus application presents only one module at a time to the user. Presenting only one module makes the interface less complicated because the interface shows fewer GUI controls and allows the user to focus on one major task at a time. Abaqus is designed to manipulate modules by swapping in one module's GUI while swapping out the previous module's GUI when requested by the user.

Toolsets are similar to modules in that they group functionality into logical units. However, toolsets generally contain less functionality than modules because toolsets focus on one particular task; for example, partitioning. Toolsets can be used in more than one module.

The Abaqus/CAE main window

This section describes how you can create an application by deriving a new class from the AFXMainWindow class and registering the modules and toolsets used by your application.

In this section:

- *Main window example*
- *Importing modules*
- *Constructing the base class*
- *Registering persistent toolsets*
- *Registering modules*

Main window example

To create a main window for a particular application, you start by deriving a new class from the AFXMainWindow class.
In the constructor of the main window, you register the modules and toolsets used by your application.

The following script constructs the Abaqus/CAE main window. The script is described in detail in the following sections. Details of how you construct modules and toolsets are given in *Creating a GUI module* and *Creating a GUI toolset*.

```
from abaqusGui import *
class CaeMainWindow(AFXMainWindow):
     def __init__(self, app, windowTitle=''):
         # Construct the GUI infrastructure.
         #
         AFXMainWindow.__init__(self, app, windowTitle)

         # Register the "persistent" toolsets.
         #
         self.registerToolset(FileToolsetGui(),
             GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
         self.registerToolset(ModelToolsetGui(),
             GUI_IN_MENUBAR)
         self.registerToolset(CanvasToolsetGui(),
             GUI_IN_MENUBAR)
         self.registerToolset(ViewManipToolsetGui(),
             GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
         self.registerToolset(TreeToolsetGui(),
             GUI_IN_MENUBAR)
         self.registerToolset(AnnotationToolsetGui(),
             GUI_IN_MENUBAR|GUI_IN_TOOLBAR)
         self.registerToolset(CustomizeToolsetGui(),
             GUI_IN_TOOL_PANE)
         self.registerToolset(SelectionToolsetGui(),
             GUI_IN_TOOLBAR)
         registerPluginToolset()
         self.registerHelpToolset(HelpToolsetGui(),
             GUI_IN_MENUBAR|GUI_IN_TOOLBAR)

         # Register modules.
         #
         self.registerModule('Part',          'Part')
         self.registerModule('Property',  'Property')
         self.registerModule('Assembly',      'Assembly')
         self.registerModule('Step',          'Step')
         self.registerModule('Interaction', 'Interaction')
         self.registerModule('Load',          'Load')
         self.registerModule('Mesh',          'Mesh')
         self.registerModule('Job', 'Job')
         self.registerModule('Visualization', 'Visualization')
         self.registerModule('Sketch',        'Sketch')
```

Importing modules

The `abaqusGui` module provides access to the entire Abaqus GUI Toolkit in addition to the modules, such as `FileToolsetGui`, that must be registered with the main window.

Constructing the base class

The first statement in the `CaeMainWindow` constructor initializes the class by calling the base class constructor.

In general, you should always call the base class constructor of the class from which you are deriving, unless you know that you will overwrite the functionality of the class.

Registering persistent toolsets

Toolsets that are registered with the main window, as opposed to being registered with a module, are available in the GUI when the application first starts up.
In addition, toolsets that are registered with the main window remain available throughout a session as the user switches modules.

To register a toolset, you call the `registerToolset` method and pass in an instance of the toolset class. You can register a help toolset with the application using the `registerHelpToolset` method. A toolset that is registered in this manner always appears to the right of all other menus in the menu bar. For more information, see *Registering toolsets*.



Note: Every application must register `viewManipToolsetGui`.

Registering modules

Registering modules puts the module names into the **Module** combo box in the context bar.
The order in which the modules are registered is the order in which the modules will appear in the **Module** combo box in the context bar.

To register a module, you call the `registerModule` method. The `registerModule` method takes the following arguments:

displayedName

A string that the application will display in the **Module** combo box in the context bar.

moduleImportName

A string that specifies the name of the module to be imported. It is your responsibility to ensure that this name is the same as your GUI module file name (without the .py extension). For more information, see *Instantiating the GUI module*.

kernelInitializationCommand

A string that specifies the name of the Python command sent to the kernel when the module is loaded.

Icons

The Abaqus GUI Toolkit supports the following formats for creating icons: XPM, BMP, GIF, and PNG.

You can use most image editing programs to produce icon images in one of the supported formats. After you have created the image file, you construct the icon by calling the appropriate method, as shown in the following example:

```
icon = afxCreatePNGIcon('myIcon.png')
FXLabel(self, 'A label with an icon', icon)
```

In some cases you may need to call the icon's `create` method before using it in a widget. In the previous example, it is not necessary to call the icon's `create` method because the label widget creates the icon when the label is created. However, if you construct an icon after you call the widget's `create` method, you must call the icon's `create` method before you use the icon in the widget. For more information, see *The create method*.

The format of an XPM icon is simple; and you can use any pixmap editor, or even a text editor, to create the icon data. More details about the XPM format are available at *https://www.x.org/docs/XPM/xpm.pdf*. The following image editing programs support the XPM format:

- ImageMagick (*http://www.imagemagick.org*)
- The GIMP (*http://www.gimp.org*)

As an alternative to using the `afxCreateXPMIcon` method, you can define the XPM image data as a Python list of strings and create an icon using the `FXXPMIcon` method, as shown in the following example.



Note: For a list of valid color names and their corresponding RGB values, see *Colors and RGB values*. To define a transparent color, you must define it as "`c None s None`", not just "`c None`".

```
blueIconData = [
"12 12 2 1",
".  c None  s None",
"   c blue",
"            ",
"            ",
"            ",
"            ",
"    ....    ",
"    ....    ",
"    ....    ",
"    ....    ",
"            ",
"            ",
"            ",
"            "
]
blueIcon = FXXPMIcon(getAFXApp(),blueIconData)
```

Figure 1 shows the blue square icon created by this example.



Figure 1: The blue square icon.

Colors and RGB values

When you are specifying a color, some methods require a string and other methods require an FXColor. To create an FXColor, you use the FXRGB function and pass in the appropriate values for red, green, and blue.

The following table shows a list of valid color strings and the corresponding RGB values.

String	RGB value
AliceBlue	FXRGB(240,248,255)
AntiqueWhite	FXRGB(250,235,215)
AntiqueWhite1	FXRGB(255,239,219)
AntiqueWhite2	FXRGB(238,223,204)
AntiqueWhite3	FXRGB(205,192,176)
AntiqueWhite4	FXRGB(139,131,120)
Aquamarine	FXRGB(127,255,212)
Aquamarine1q	FXRGB(127,255,212)
Aquamarine2	FXRGB(118,238,198)
Aquamarine3	FXRGB(102,205,170)
Aquamarine4	FXRGB(69,139,116)
Azure	FXRGB(240,255,255)
Azure1	FXRGB(240,255,255)
Azure2	FXRGB(224,238,238)
Azure3	FXRGB(193,205,205)
Azure4	FXRGB(131,139,139)
Beige	FXRGB(245,245,220)
Bisque	FXRGB(255,228,196)
Bisque1	FXRGB(255,228,196)
Bisque2	FXRGB(238,213,183)
Bisque3	FXRGB(205,183,158)
Bisque4	FXRGB(139,125,107)
Black	FXRGB(0,0,0)
BlanchedAlmond	FXRGB(255,235,205)
Blue	FXRGB(0,0,255)
Blue1	FXRGB(0,0,255)
Blue2	FXRGB(0,0,238)
Blue3	FXRGB(0,0,205)
Blue4	FXRGB(0,0,139)
BlueViolet	FXRGB(138, 43,226)
Brown	FXRGB(165, 42, 42)
Brown1	FXRGB(255, 64, 64)

String	RGB value
Brown2	FXRGB(238, 59, 59)
Brown3	FXRGB(205, 51, 51)
Brown4	FXRGB(139, 35, 35)
Burlywood	FXRGB(222,184,135)
Burlywood1	FXRGB(255,211,155)
Burlywood2	FXRGB(238,197,145)
Burlywood3	FXRGB(205,170,125)
Burlywood4	FXRGB(139,115, 85)
CadetBlue	FXRGB(95,158,160)
CadetBlue1	FXRGB(152,245,255)
CadetBlue2	FXRGB(142,229,238)
CadetBlue3	FXRGB(122,197,205)
CadetBlue4	FXRGB(83,134,139)
Chartreuse	FXRGB(127,255,0)
Chartreuse1	FXRGB(127,255,0)
Chartreuse2	FXRGB(118,238,0)
Chartreuse3	FXRGB(102,205,0)
Chartreuse4	FXRGB(69,139,0)
Chocolate	FXRGB(210,105, 30)
Chocolate1	FXRGB(255,127, 36)
Chocolate2	FXRGB(238,118, 33)
Chocolate3	FXRGB(205,102, 29)
Chocolate4	FXRGB(139, 69, 19)
Coral	FXRGB(255,127, 80)
Coral1	FXRGB(255,114, 86)
Coral2	FXRGB(238,106, 80)
Coral3	FXRGB(205, 91, 69)
Coral4	FXRGB(139, 62, 47)
CornflowerBlue	FXRGB(100,149,237)
Cornsilk	FXRGB(255,248,220)
Cornsilk1	FXRGB(255,248,220)
Cornsilk2	FXRGB(238,232,205)
Cornsilk3	FXRGB(205,200,177)
Cornsilk4	FXRGB(139,136,120)
Cyan	FXRGB(0,255,255)
Cyan1	FXRGB(0,255,255)
Cyan2	FXRGB(0,238,238)
Cyan3	FXRGB(0,205,205)
Cyan4	FXRGB(0,139,139)

String	RGB value
DarkBlue	FXRGB(0,0,139)
DarkCyan	FXRGB(0,139,139)
DarkGoldenrod	FXRGB(184,134, 11)
DarkGoldenrod1	FXRGB(255,185, 15)
DarkGoldenrod2	FXRGB(238,173, 14)
DarkGoldenrod3	FXRGB(205,149, 12)
DarkGoldenrod4	FXRGB(139,101,8)
DarkGray	FXRGB(169,169,169)
DarkGreen	FXRGB(0,100,0)
DarkGrey	FXRGB(169,169,169)
DarkKhaki	FXRGB(189,183,107)
DarkMagenta	FXRGB(139,0,139)
DarkOliveGreen	FXRGB(85,107, 47)
DarkOliveGreen1	FXRGB(202,255,112)
DarkOliveGreen2	FXRGB(188,238,104)
DarkOliveGreen3	FXRGB(162,205, 90)
DarkOliveGreen4	FXRGB(110,139, 61)
DarkOrange	FXRGB(255,140,0)
DarkOrange1	FXRGB(255,127,0)
DarkOrange2	FXRGB(238,118,0)
DarkOrange3	FXRGB(205,102,0)
DarkOrange4	FXRGB(139, 69,0)
DarkOrchid	FXRGB(153, 50,204)
DarkOrchid1	FXRGB(191, 62,255)
DarkOrchid2	FXRGB(178, 58,238)
DarkOrchid3	FXRGB(154, 50,205)
DarkOrchid4	FXRGB(104, 34,139)
DarkRed	FXRGB(139,0,0)
DarkSalmon	FXRGB(233,150,122)
DarkSeaGreen	FXRGB(143,188,143)
DarkSeaGreen1	FXRGB(193,255,193)
DarkSeaGreen2	FXRGB(180,238,180)
DarkSeaGreen3	FXRGB(155,205,155)
DarkSeaGreen4	FXRGB(105,139,105)
DarkSlateBlue	FXRGB(72, 61,139)
DarkSlateGray	FXRGB(47, 79, 79)
DarkSlateGray1	FXRGB(151,255,255)
DarkSlateGray2	FXRGB(141,238,238)
DarkSlateGray3	FXRGB(121,205,205)

String	RGB value
DarkSlateGray4	FXRGB(82,139,139)
DarkSlateGrey	FXRGB(47, 79, 79)
DarkTurquoise	FXRGB(0,206,209)
DarkViolet	FXRGB(148,0,211)
DeepPink	FXRGB(255, 20,147)
DeepPink1	FXRGB(255, 20,147)
DeepPink2	FXRGB(238, 18,137)
DeepPink3	FXRGB(205, 16,118)
DeepPink4	FXRGB(139, 10, 80)
DeepSkyBlue	FXRGB(0,191,255)
DeepSkyBlue1	FXRGB(0,191,255)
DeepSkyBlue2	FXRGB(0,178,238)
DeepSkyBlue3	FXRGB(0,154,205)
DeepSkyBlue4	FXRGB(0,104,139)
DimGray	FXRGB(105,105,105)
DimGrey	FXRGB(105,105,105)
DodgerBlue	FXRGB(30,144,255)
DodgerBlue1	FXRGB(30,144,255)
DodgerBlue2	FXRGB(28,134,238)
DodgerBlue3	FXRGB(24,116,205)
DodgerBlue4	FXRGB(16, 78,139)
Firebrick	FXRGB(178, 34, 34)
Firebrick1	FXRGB(255, 48, 48)
Firebrick2	FXRGB(238, 44, 44)
Firebrick3	FXRGB(205, 38, 38)
Firebrick4	FXRGB(139, 26, 26)
FloralWhite	FXRGB(255,250,240)
ForestGreen	FXRGB(34,139, 34)
Gainsboro	FXRGB(220,220,220)
GhostWhite	FXRGB(248,248,255)
Gold	FXRGB(255,215,0)
Gold1	FXRGB(255,215,0)
Gold2	FXRGB(238,201,0)
Gold3	FXRGB(205,173,0)
Gold4	FXRGB(139,117,0)
Goldenrod	FXRGB(218,165, 32)
Goldenrod1	FXRGB(255,193, 37)
Goldenrod2	FXRGB(238,180, 34)
Goldenrod3	FXRGB(205,155, 29)

String	RGB value
Goldenrod4	FXRGB(139,105, 20)
Gray	FXRGB(190,190,190)
Gray0	FXRGB(0,0,0)
Gray1	FXRGB(3,3,3)
Gray10	FXRGB(26, 26, 26)
Gray100	FXRGB(255,255,255)
Gray11	FXRGB(28, 28, 28)
Gray12	FXRGB(31, 31, 31)
Gray13	FXRGB(33, 33, 33)
Gray14	FXRGB(36, 36, 36)
Gray15	FXRGB(38, 38, 38)
Gray16	FXRGB(41, 41, 41)
Gray17	FXRGB(43, 43, 43)
Gray18	FXRGB(46, 46, 46)
Gray19	FXRGB(48, 48, 48)
Gray2	FXRGB(5,5,5)
Gray20	FXRGB(51, 51, 51)
Gray21	FXRGB(54, 54, 54)
Gray22	FXRGB(56, 56, 56)
Gray23	FXRGB(59, 59, 59)
Gray24	FXRGB(61, 61, 61)
Gray25	FXRGB(64, 64, 64)
Gray26	FXRGB(66, 66, 66)
Gray27	FXRGB(69, 69, 69)
Gray28	FXRGB(71, 71, 71)
Gray29	FXRGB(74, 74, 74)
Gray3	FXRGB(8,8,8)
Gray30	FXRGB(77, 77, 77)
Gray31	FXRGB(79, 79, 79)
Gray32	FXRGB(82, 82, 82)
Gray33	FXRGB(84, 84, 84)
Gray34	FXRGB(87, 87, 87)
Gray35	FXRGB(89, 89, 89)
Gray36	FXRGB(92, 92, 92)
Gray37	FXRGB(94, 94, 94)
Gray38	FXRGB(97, 97, 97)
Gray39	FXRGB(99, 99, 99)
Gray4	FXRGB(10, 10, 10)
Gray40	FXRGB(102,102,102)

String	RGB value
Gray41	FXRGB(105,105,105)
Gray42	FXRGB(107,107,107)
Gray43	FXRGB(110,110,110)
Gray44	FXRGB(112,112,112)
Gray45	FXRGB(115,115,115)
Gray46	FXRGB(117,117,117)
Gray47	FXRGB(120,120,120)
Gray48	FXRGB(122,122,122)
Gray49	FXRGB(125,125,125)
Gray5	FXRGB(13, 13, 13)
Gray50	FXRGB(127,127,127)
Gray51	FXRGB(130,130,130)
Gray52	FXRGB(133,133,133)
Gray53	FXRGB(135,135,135)
Gray54	FXRGB(138,138,138)
Gray55	FXRGB(140,140,140)
Gray56	FXRGB(143,143,143)
Gray57	FXRGB(145,145,145)
Gray58	FXRGB(148,148,148)
Gray59	FXRGB(150,150,150)
Gray6	FXRGB(15, 15, 15)
Gray60	FXRGB(153,153,153)
Gray61	FXRGB(156,156,156)
Gray62	FXRGB(158,158,158)
Gray63	FXRGB(161,161,161)
Gray64	FXRGB(163,163,163)
Gray65	FXRGB(166,166,166)
Gray66	FXRGB(168,168,168)
Gray67	FXRGB(171,171,171)
Gray68	FXRGB(173,173,173)
Gray69	FXRGB(176,176,176)
Gray7	FXRGB(18, 18, 18)
Gray70	FXRGB(179,179,179)
Gray71	FXRGB(181,181,181)
Gray72	FXRGB(184,184,184)
Gray73	FXRGB(186,186,186)
Gray74	FXRGB(189,189,189)
Gray75	FXRGB(191,191,191)
Gray76	FXRGB(194,194,194)

String	RGB value
Gray77	FXRGB(196,196,196)
Gray78	FXRGB(199,199,199)
Gray79	FXRGB(201,201,201)
Gray8	FXRGB(20, 20, 20)
Gray80	FXRGB(204,204,204)
Gray81	FXRGB(207,207,207)
Gray82	FXRGB(209,209,209)
Gray83	FXRGB(212,212,212)
Gray84	FXRGB(214,214,214)
Gray85	FXRGB(217,217,217)
Gray86	FXRGB(219,219,219)
Gray87	FXRGB(222,222,222)
Gray88	FXRGB(224,224,224)
Gray89	FXRGB(227,227,227)
Gray9	FXRGB(23, 23, 23)
Gray90	FXRGB(229,229,229)
Gray91	FXRGB(232,232,232)
Gray92	FXRGB(235,235,235)
Gray93	FXRGB(237,237,237)
Gray94	FXRGB(240,240,240)
Gray95	FXRGB(242,242,242)
Gray96	FXRGB(245,245,245)
Gray97	FXRGB(247,247,247)
Gray98	FXRGB(250,250,250)
Gray99	FXRGB(252,252,252)
Green	FXRGB(0,255,0)
Green1	FXRGB(0,255,0)
Green2	FXRGB(0,238,0)
Green3	FXRGB(0,205,0)
Green4	FXRGB(0,139,0)
GreenYellow	FXRGB(173,255, 47)
Grey	FXRGB(190,190,190)
Grey0	FXRGB(0,0,0)
Grey1	FXRGB(3,3,3)
Grey10	FXRGB(26, 26, 26)
Grey100	FXRGB(255,255,255)
Grey11	FXRGB(28, 28, 28)
Grey12	FXRGB(31, 31, 31)
Grey13	FXRGB(33, 33, 33)

String	RGB value
Grey14	FXRGB(36, 36, 36)
Grey15	FXRGB(38, 38, 38)
Grey16	FXRGB(41, 41, 41)
Grey17	FXRGB(43, 43, 43)
Grey18	FXRGB(46, 46, 46)
Grey19	FXRGB(48, 48, 48)
Grey2	FXRGB(5,5,5)
Grey20	FXRGB(51, 51, 51)
Grey21	FXRGB(54, 54, 54)
Grey22	FXRGB(56, 56, 56)
Grey23	FXRGB(59, 59, 59)
Grey24	FXRGB(61, 61, 61)
Grey25	FXRGB(64, 64, 64)
Grey26	FXRGB(66, 66, 66)
Grey27	FXRGB(69, 69, 69)
Grey28	FXRGB(71, 71, 71)
Grey29	FXRGB(74, 74, 74)
Grey3	FXRGB(8,8,8)
Grey30	FXRGB(77, 77, 77)
Grey31	FXRGB(79, 79, 79)
Grey32	FXRGB(82, 82, 82)
Grey33	FXRGB(84, 84, 84)
Grey34	FXRGB(87, 87, 87)
Grey35	FXRGB(89, 89, 89)
Grey36	FXRGB(92, 92, 92)
Grey37	FXRGB(94, 94, 94)
Grey38	FXRGB(97, 97, 97)
Grey39	FXRGB(99, 99, 99)
Grey4	FXRGB(10, 10, 10)
Grey40	FXRGB(102,102,102)
Grey41	FXRGB(105,105,105)
Grey42	FXRGB(107,107,107)
Grey43	FXRGB(110,110,110)
Grey44	FXRGB(112,112,112)
Grey45	FXRGB(115,115,115)
Grey46	FXRGB(117,117,117)
Grey47	FXRGB(120,120,120)
Grey48	FXRGB(122,122,122)
Grey49	FXRGB(125,125,125)

String	RGB value
Grey5	FXRGB(13, 13, 13)
Grey50	FXRGB(127,127,127)
Grey51	FXRGB(130,130,130)
Grey52	FXRGB(133,133,133)
Grey53	FXRGB(135,135,135)
Grey54	FXRGB(138,138,138)
Grey55	FXRGB(140,140,140)
Grey56	FXRGB(143,143,143)
Grey57	FXRGB(145,145,145)
Grey58	FXRGB(148,148,148)
Grey59	FXRGB(150,150,150)
Grey6	FXRGB(15, 15, 15)
Grey60	FXRGB(153,153,153)
Grey61	FXRGB(156,156,156)
Grey62	FXRGB(158,158,158)
Grey63	FXRGB(161,161,161)
Grey64	FXRGB(163,163,163)
Grey65	FXRGB(166,166,166)
Grey66	FXRGB(168,168,168)
Grey67	FXRGB(171,171,171)
Grey68	FXRGB(173,173,173)
Grey69	FXRGB(176,176,176)
Grey7	FXRGB(18, 18, 18)
Grey70	FXRGB(179,179,179)
Grey71	FXRGB(181,181,181)
Grey72	FXRGB(184,184,184)
Grey73	FXRGB(186,186,186)
Grey74	FXRGB(189,189,189)
Grey75	FXRGB(191,191,191)
Grey76	FXRGB(194,194,194)
Grey77	FXRGB(196,196,196)
Grey78	FXRGB(199,199,199)
Grey79	FXRGB(201,201,201)
Grey8	FXRGB(20, 20, 20)
Grey80	FXRGB(204,204,204)
Grey81	FXRGB(207,207,207)
Grey82	FXRGB(209,209,209)
Grey83	FXRGB(212,212,212)
Grey84	FXRGB(214,214,214)

String	RGB value
Grey85	FXRGB(217,217,217)
Grey86	FXRGB(219,219,219)
Grey87	FXRGB(222,222,222)
Grey88	FXRGB(224,224,224)
Grey89	FXRGB(227,227,227)
Grey9	FXRGB(23, 23, 23)
Grey90	FXRGB(229,229,229)
Grey91	FXRGB(232,232,232)
Grey92	FXRGB(235,235,235)
Grey93	FXRGB(237,237,237)
Grey94	FXRGB(240,240,240)
Grey95	FXRGB(242,242,242)
Grey96	FXRGB(245,245,245)
Grey97	FXRGB(247,247,247)
Grey98	FXRGB(250,250,250)
Grey99	FXRGB(252,252,252)
Honeydew	FXRGB(240,255,240)
Honeydew1	FXRGB(240,255,240)
Honeydew2	FXRGB(224,238,224)
Honeydew3	FXRGB(193,205,193)
Honeydew4	FXRGB(131,139,131)
HotPink	FXRGB(255,105,180)
HotPink1	FXRGB(255,110,180)
HotPink2	FXRGB(238,106,167)
HotPink3	FXRGB(205, 96,144)
HotPink4	FXRGB(139, 58, 98)
IndianRed	FXRGB(205, 92, 92)
IndianRed1	FXRGB(255,106,106)
IndianRed2	FXRGB(238, 99, 99)
IndianRed3	FXRGB(205, 85, 85)
IndianRed4	FXRGB(139, 58, 58)
Ivory	FXRGB(255,255,240)
Ivory1	FXRGB(255,255,240)
Ivory2	FXRGB(238,238,224)
Ivory3	FXRGB(205,205,193)
Ivory4	FXRGB(139,139,131)
Khaki	FXRGB(240,230,140)
Khaki1	FXRGB(255,246,143)
Khaki2	FXRGB(238,230,133)

String	RGB value
Khaki3	FXRGB(205,198,115)
Khaki4	FXRGB(139,134, 78)
Lavender	FXRGB(230,230,250)
LavenderBlush	FXRGB(255,240,245)
LavenderBlush1	FXRGB(255,240,245)
LavenderBlush2	FXRGB(238,224,229)
LavenderBlush3	FXRGB(205,193,197)
LavenderBlush4	FXRGB(139,131,134)
LawnGreen	FXRGB(124,252,0)
LemonChiffon	FXRGB(255,250,205)
LemonChiffon1	FXRGB(255,250,205)
LemonChiffon2	FXRGB(238,233,191)
LemonChiffon3	FXRGB(205,201,165)
LemonChiffon4	FXRGB(139,137,112)
LightBlue	FXRGB(173,216,230)
LightBlue1	FXRGB(191,239,255)
LightBlue2	FXRGB(178,223,238)
LightBlue3	FXRGB(154,192,205)
LightBlue4	FXRGB(104,131,139)
LightCoral	FXRGB(240,128,128)
LightCyan	FXRGB(224,255,255)
LightCyan1	FXRGB(224,255,255)
LightCyan2	FXRGB(209,238,238)
LightCyan3	FXRGB(180,205,205)
LightCyan4	FXRGB(122,139,139)
LightGoldenrod	FXRGB(238,221,130)
LightGoldenrod1	FXRGB(255,236,139)
LightGoldenrod2	FXRGB(238,220,130)
LightGoldenrod3	FXRGB(205,190,112)
LightGoldenrod4	FXRGB(139,129, 76)
LightGoldenrodYellow	FXRGB(250,250,210)
LightGray	FXRGB(211,211,211)
LightGreen	FXRGB(144,238,144)
LightGrey	FXRGB(211,211,211)
LightPink	FXRGB(255,182,193)
LightPink1	FXRGB(255,174,185)
LightPink2	FXRGB(238,162,173)
LightPink3	FXRGB(205,140,149)
LightPink4	FXRGB(139, 95,101)

String	RGB value
LightSalmon	FXRGB(255,160,122)
LightSalmon1	FXRGB(255,160,122)
LightSalmon2	FXRGB(238,149,114)
LightSalmon3	FXRGB(205,129, 98)
LightSalmon4	FXRGB(139, 87, 66)
LightSeaGreen	FXRGB(32,178,170)
LightSkyBlue	FXRGB(135,206,250)
LightSkyBlue1	FXRGB(176,226,255)
LightSkyBlue2	FXRGB(164,211,238)
LightSkyBlue3	FXRGB(141,182,205)
LightSkyBlue4	FXRGB(96,123,139)
LightSlateBlue	FXRGB(132,112,255)
LightSlateGray	FXRGB(119,136,153)
LightSlateGrey	FXRGB(119,136,153)
LightSteelBlue	FXRGB(176,196,222)
LightSteelBlue1	FXRGB(202,225,255)
LightSteelBlue2	FXRGB(188,210,238)
LightSteelBlue3	FXRGB(162,181,205)
LightSteelBlue4	FXRGB(110,123,139)
LightYellow	FXRGB(255,255,224)
LightYellow1	FXRGB(255,255,224)
LightYellow2	FXRGB(238,238,209)
LightYellow3	FXRGB(205,205,180)
LightYellow4	FXRGB(139,139,122)
LimeGreen	FXRGB(50,205, 50)
Linen	FXRGB(250,240,230)
Magenta	FXRGB(255,0,255)
Magenta1	FXRGB(255,0,255)
Magenta2	FXRGB(238,0,238)
Magenta3	FXRGB(205,0,205)
Magenta4	FXRGB(139,0,139)
Maroon	FXRGB(176, 48, 96)
Maroon1	FXRGB(255, 52,179)
Maroon2	FXRGB(238, 48,167)
Maroon3	FXRGB(205, 41,144)
Maroon4	FXRGB(139, 28, 98)
MediumAquamarine	FXRGB(102,205,170)
MediumBlue	FXRGB(0,0,205)
MediumOrchid	FXRGB(186, 85,211)

String	RGB value
MediumOrchid1	FXRGB(224,102,255)
MediumOrchid2	FXRGB(209, 95,238)
MediumOrchid3	FXRGB(180, 82,205)
MediumOrchid4	FXRGB(122, 55,139)
MediumPurple	FXRGB(147,112,219)
MediumPurple1	FXRGB(171,130,255)
MediumPurple2	FXRGB(159,121,238)
MediumPurple3	FXRGB(137,104,205)
MediumPurple4	FXRGB(93, 71,139)
MediumSeaGreen	FXRGB(60,179,113)
MediumSlateBlue	FXRGB(123,104,238)
MediumSpringGreen	FXRGB(0,250,154)
MediumTurquoise	FXRGB(72,209,204)
MediumVioletRed	FXRGB(199, 21,133)
MidnightBlue	FXRGB(25, 25,112)
MintCream	FXRGB(245,255,250)
MistyRose	FXRGB(255,228,225)
MistyRose1	FXRGB(255,228,225)
MistyRose2	FXRGB(238,213,210)
MistyRose3	FXRGB(205,183,181)
MistyRose4	FXRGB(139,125,123)
Moccasin	FXRGB(255,228,181)
NavajoWhite	FXRGB(255,222,173)
NavajoWhite1	FXRGB(255,222,173)
NavajoWhite2	FXRGB(238,207,161)
NavajoWhite3	FXRGB(205,179,139)
NavajoWhite4	FXRGB(139,121, 94)
Navy	FXRGB(0,0,128)
NavyBlue	FXRGB(0,0,128)
None	FXRGB(0,0,0,0)
OldLace	FXRGB(253,245,230)
OliveDrab	FXRGB(107,142, 35)
OliveDrab1	FXRGB(192,255, 62)
OliveDrab2	FXRGB(179,238, 58)
OliveDrab3	FXRGB(154,205, 50)
OliveDrab4	FXRGB(105,139, 34)
Orange	FXRGB(255,165,0)
Orange1	FXRGB(255,165,0)
Orange2	FXRGB(238,154,0)

String	RGB value
Orange3	FXRGB(205,133,0)
Orange4	FXRGB(139, 90,0)
OrangeRed	FXRGB(255, 69,0)
OrangeRed1	FXRGB(255, 69,0)
OrangeRed2	FXRGB(238, 64,0)
OrangeRed3	FXRGB(205, 55,0)
OrangeRed4	FXRGB(139, 37,0)
Orchid	FXRGB(218,112,214)
Orchid1	FXRGB(255,131,250)
Orchid2	FXRGB(238,122,233)
Orchid3	FXRGB(205,105,201)
Orchid4	FXRGB(139, 71,137)
PaleGoldenrod	FXRGB(238,232,170)
PaleGreen	FXRGB(152,251,152)
PaleGreen1	FXRGB(154,255,154)
PaleGreen2	FXRGB(144,238,144)
PaleGreen3	FXRGB(124,205,124)
PaleGreen4	FXRGB(84,139, 84)
PaleTurquoise	FXRGB(175,238,238)
PaleTurquoise1	FXRGB(187,255,255)
PaleTurquoise2	FXRGB(174,238,238)
PaleTurquoise3	FXRGB(150,205,205)
PaleTurquoise4	FXRGB(102,139,139)
PaleVioletRed	FXRGB(219,112,147)
PaleVioletRed1	FXRGB(255,130,171)
PaleVioletRed2	FXRGB(238,121,159)
PaleVioletRed3	FXRGB(205,104,137)
PaleVioletRed4	FXRGB(139, 71, 93)
PapayaWhip	FXRGB(255,239,213)
PeachPuff	FXRGB(255,218,185)
PeachPuff1	FXRGB(255,218,185)
PeachPuff2	FXRGB(238,203,173)
PeachPuff3	FXRGB(205,175,149)
PeachPuff4	FXRGB(139,119,101)
Peru	FXRGB(205,133, 63)
Pink	FXRGB(255,192,203)
Pink1	FXRGB(255,181,197)
Pink2	FXRGB(238,169,184)
Pink3	FXRGB(205,145,158)

String	RGB value
Pink4	FXRGB(139, 99,108)
Plum	FXRGB(221,160,221)
Plum1	FXRGB(255,187,255)
Plum2	FXRGB(238,174,238)
Plum3	FXRGB(205,150,205)
Plum4	FXRGB(139,102,139)
PowderBlue	FXRGB(176,224,230)
Purple	FXRGB(160, 32,240)
Purple1	FXRGB(155, 48,255)
Purple2	FXRGB(145, 44,238)
Purple3	FXRGB(125, 38,205)
Purple4	FXRGB(85, 26,139)
Red	FXRGB(255,0,0)
Red1	FXRGB(255,0,0)
Red2	FXRGB(238,0,0)
Red3	FXRGB(205,0,0)
Red4	FXRGB(139,0,0)
RosyBrown	FXRGB(188,143,143)
RosyBrown1	FXRGB(255,193,193)
RosyBrown2	FXRGB(238,180,180)
RosyBrown3	FXRGB(205,155,155)
RosyBrown4	FXRGB(139,105,105)
RoyalBlue	FXRGB(65,105,225)
RoyalBlue1	FXRGB(72,118,255)
RoyalBlue2	FXRGB(67,110,238)
RoyalBlue3	FXRGB(58, 95,205)
RoyalBlue4	FXRGB(39, 64,139)
SaddleBrown	FXRGB(139, 69, 19)
Salmon	FXRGB(250,128,114)
Salmon1	FXRGB(255,140,105)
Salmon2	FXRGB(238,130, 98)
Salmon3	FXRGB(205,112, 84)
Salmon4	FXRGB(139, 76, 57)
SandyBrown	FXRGB(244,164, 96)
SeaGreen	FXRGB(46,139, 87)
SeaGreen1	FXRGB(84,255,159)
SeaGreen2	FXRGB(78,238,148)
SeaGreen3	FXRGB(67,205,128)
SeaGreen4	FXRGB(46,139, 87)

String	RGB value
Seashell	FXRGB(255,245,238)
Seashell1	FXRGB(255,245,238)
Seashell2	FXRGB(238,229,222)
Seashell3	FXRGB(205,197,191)
Seashell4	FXRGB(139,134,130)
Sienna	FXRGB(160, 82, 45)
Sienna1	FXRGB(255,130, 71)
Sienna2	FXRGB(238,121, 66)
Sienna3	FXRGB(205,104, 57)
Sienna4	FXRGB(139, 71, 38)
SkyBlue	FXRGB(135,206,235)
SkyBlue1	FXRGB(135,206,255)
SkyBlue2	FXRGB(126,192,238)
SkyBlue3	FXRGB(108,166,205)
SkyBlue4	FXRGB(74,112,139)
SlateBlue	FXRGB(106, 90,205)
SlateBlue1	FXRGB(131,111,255)
SlateBlue2	FXRGB(122,103,238)
SlateBlue3	FXRGB(105, 89,205)
SlateBlue4	FXRGB(71, 60,139)
SlateGray	FXRGB(112,128,144)
SlateGray1	FXRGB(198,226,255)
SlateGray2	FXRGB(185,211,238)
SlateGray3	FXRGB(159,182,205)
SlateGray4	FXRGB(108,123,139)
SlateGrey	FXRGB(112,128,144)
Snow	FXRGB(255,250,250)
Snow1	FXRGB(255,250,250)
Snow2	FXRGB(238,233,233)
Snow3	FXRGB(205,201,201)
Snow4	FXRGB(139,137,137)
SpringGreen	FXRGB(0,255,127)
SpringGreen1	FXRGB(0,255,127)
SpringGreen2	FXRGB(0,238,118)
SpringGreen3	FXRGB(0,205,102)
SpringGreen4	FXRGB(0,139, 69)
SteelBlue	FXRGB(70,130,180)
SteelBlue1	FXRGB(99,184,255)
SteelBlue2	FXRGB(92,172,238)

String	RGB value
SteelBlue3	FXRGB(79,148,205)
SteelBlue4	FXRGB(54,100,139)
Tan	FXRGB(210,180,140)
Tan1	FXRGB(255,165, 79)
Tan2	FXRGB(238,154, 73)
Tan3	FXRGB(205,133, 63)
Tan4	FXRGB(139, 90, 43)
Thistle	FXRGB(216,191,216)
Thistle1	FXRGB(255,225,255)
Thistle2	FXRGB(238,210,238)
Thistle3	FXRGB(205,181,205)
Thistle4	FXRGB(139,123,139)
Tomato	FXRGB(255, 99, 71)
Tomato1	FXRGB(255, 99, 71)
Tomato2	FXRGB(238, 92, 66)
Tomato3	FXRGB(205, 79, 57)
Tomato4	FXRGB(139, 54, 38)
Turquoise	FXRGB(64,224,208)
Turquoise1	FXRGB(0,245,255)
Turquoise2	FXRGB(0,229,238)
Turquoise3	FXRGB(0,197,205)
Turquoise4	FXRGB(0,134,139)
Violet	FXRGB(238,130,238)
VioletRed	FXRGB(208, 32,144)
VioletRed1	FXRGB(255, 62,150)
VioletRed2	FXRGB(238, 58,140)
VioletRed3	FXRGB(205, 50,120)
VioletRed4	FXRGB(139, 34, 82)
Wheat	FXRGB(245,222,179)
Wheat1	FXRGB(255,231,186)
Wheat2	FXRGB(238,216,174)
Wheat3	FXRGB(205,186,150)
Wheat4	FXRGB(139,126,102)
White	FXRGB(255,255,255)
WhiteSmoke	FXRGB(245,245,245)
Yellow	FXRGB(255,255,0)
Yellow1	FXRGB(255,255,0)
Yellow2	FXRGB(238,238,0)
Yellow3	FXRGB(205,205,0)

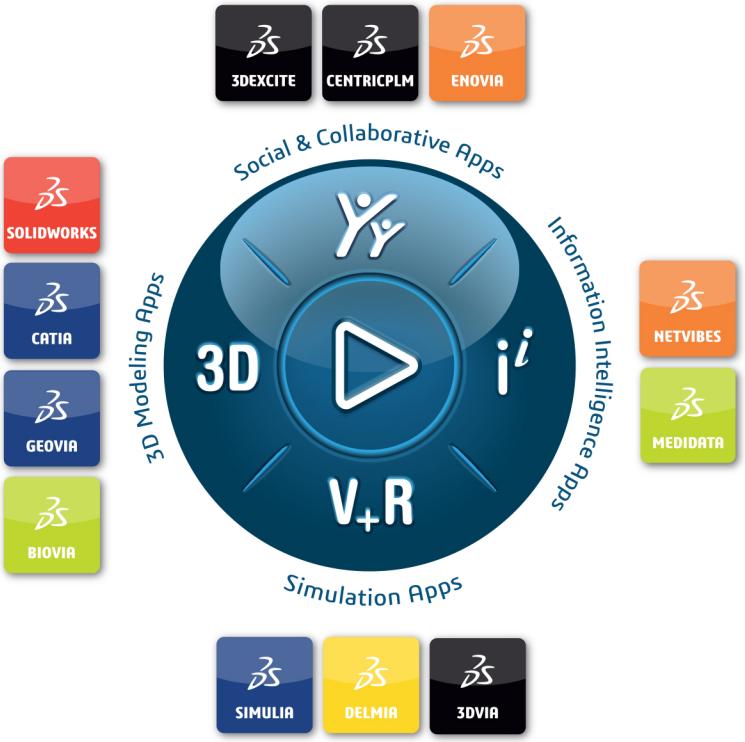
String	RGB value
Yellow4	FXRGB(139,139,0)
YellowGreen	FXRGB(154,205, 50)

Layout hints

The layout managers in the Abaqus GUI Toolkit support the layout hints.

Supported layout hints follow:

Layout hint	Used in	Effect
LAYOUT_SIDE_TOP (default) LAYOUT_SIDE_BOTTOM LAYOUT_SIDE_LEFT LAYOUT_SIDE_RIGHT	FXPacker FXGroupBox FXTopLevel	If you specify one of these four layout hints, the child widget will be stuck to the top, bottom, left, or right, respectively, in the layout manager cavity. The size of the cavity will be reduced by the amount lopped off by the packed widget. LAYOUT_SIDE_TOP and LAYOUT_SIDE_BOTTOM will reduce the height of the cavity. LAYOUT_SIDE_LEFT and LAYOUT_SIDE_RIGHT will reduce the width of the cavity. For other composite widgets, these hints may not have any effect.
LAYOUT_LEFT (default) LAYOUT_RIGHT	All	The widget will be placed on the left side or right side of the space remaining in the container. When used for a child of FXPacker FXGroupBox, or FXTopLevel, the hint will be ignored unless either LAYOUT_SIDE_TOP or LAYOUT_SIDE_BOTTOM is specified.
LAYOUT_TOP (default) LAYOUT_BOTTOM	All	The widget will be placed on the top-side or bottom-side of the space remaining in the container. For a child of FXPacker, etc., these options will only have effect if either LAYOUT_SIDE_RIGHT or LAYOUT_SIDE_LEFT is specified.
LAYOUT_CENTER_X LAYOUT_CENTER_Y	All	The widget will be centered in the *X*-direction (or *Y*-direction) in the parent. Extra spacing will be added around the widget to place it at the center of the space available to it. The size of the widget will be its default size unless LAYOUT_FIX_WIDTH or LAYOUT_FIX_HEIGHT have been specified.
LAYOUT_FILL_X LAYOUT_FILL_Y	All	Either none, one, or both of these hints may be specified. LAYOUT_FILL_X will cause the parent layout manager to stretch or shrink the widget to accommodate the available space. If more than one child with this option is placed side by side, the available space will be subdivided proportionally to their default size. LAYOUT_FILL_Y has the identical effect on the vertical direction.
LAYOUT_FIX_X LAYOUT_FIX_Y	All	Either none, one, or both of these hints may be specified. The LAYOUT_FIX_X hint will cause the parent layout manager to place this widget at the indicated *X*-position, as passed on the optional arguments in the widgets constructor argument list. Likewise, a LAYOUT_FIX_Y hint will cause placement at the indicated *Y*-position. The *X*- and *Y*-positions are specified in the parent's coordinate system.
LAYOUT_FILL_ROW LAYOUT_FILL_COLUMN	FXMatrix	If LAYOUT_FILL_COLUMN is specified for all child widgets in a certain column of a matrix layout manager, the whole column can stretch if the matrix itself is stretched horizontally. Analogously, if LAYOUT_FILL_ROW is specified for all child widgets in a certain row, the whole row is stretched if the matrix layout manager is stretched vertically.
LAYOUT_FIX_WIDTH LAYOUT_FIX_HEIGHT	All	These options will fix the widget's width (or height) to the value specified on the constructor. You can change the size of the widget using its setWidth() and setHeight() methods; however, the layout manager will generally observe the specified dimensions of the widget without trying to modify it (unless other options override).
LAYOUT_MIN_WIDTH (default) LAYOUT_MIN_HEIGHT (default)	All	Either none, one, or both of these hints may be specified. You will almost never specify these options, except perhaps for code legibility. If LAYOUT_FIX_WIDTH or LAYOUT_FIX_HEIGHT are not specified, these options will cause the parent layout widget to use the default (or minimum) width and height, respectively.



Our **3D**EXPERIENCE® platform powers our brand applications, serving 12 industries, and provides a rich portfolio of industry solution experiences.

Dassault Systèmes is a catalyst for human progress. We provide business and people with collaborative virtual environments to imagine sustainable innovations. By creating virtual twin experiences of the real world with our **3D**EXPERIENCE platform and applications, our customers can redefine the creation, production and life-cycle-management processes of their offer and thus have a meaningful impact to make the world more sustainable. The beauty of the Experience Economy is that it is a human-centered economy for the benefit of all – consumers, patients and citizens.

Dassault Systèmes brings value to more than 300,000 customers of all sizes, in all industries, in more than 150 countries. For more information, visit **www.3ds.com**.

Europe/Middle East/Africa
Dassault Systèmes
10, rue Marcel Dassault
CS 40501
78946 Vélizy-Villacoublay Cedex
France

Asia-Pacific
Dassault Systèmes
17F, Foxconn Building,
No. 1366, Lujiazui Ring Road
Pilot Free Trade Zone, Shanghai 200120
China

Americas
Dassault Systèmes
175 Wyman Street
Waltham, Massachusetts
02451-1223
USA

DASSAULT SYSTEMES

©2026 Dassault Systèmes. All rights reserved. **3DEXPERIENCE**, the 3DS logo, the Compass icon, IFWE, 3DEXCITE, 3DVIA, BIOVIA, CATIA, CENTRIC PLM, DELMIA, ENOVIA, GEOVIA, MEDIDATA, NETVIBES, OUTSCALE, SIMULIA and SOLIDWORKS are commercial trademarks or registered trademarks of Dassault Systèmes, a European company (Societas Europaea) incorporated under French law, and registered with the Versailles trade and companies registry under number 322 306 440, or its subsidiaries in the United States and/or other countries.