Ansys
2022/R2
Engineering What's Ahead.

©2022 ANSYS, Inc.
All Rights Reserved.
Unauthorized use, distribution
or duplication is prohibited.

Ansys ACT Developer's Guide

Ansys
ANSYS, Inc.
Southpointe
2600 Ansys Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
http://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2022 R2
July 2022

ANSYS, Inc. and ANSYS Europe, Ltd. are UL registered ISO 9001: 2015 companies.

Copyright and Trademark Information

© 2022 ANSYS, Inc. Unauthorized use, distribution or duplication is prohibited.

ANSYS, Ansys Workbench, AUTODYN, CFX, FLUENT and any and all ANSYS, Inc. brand, product, service and feature names, logos and slogans are registered trademarks or trademarks of ANSYS, Inc. or its subsidiaries located in the United States or other countries. ICEM CFD is a trademark used by ANSYS, Inc. under license. CFX is a trademark of Sony Corporation in Japan. All other brand, product, service and feature names or trademarks are the property of their respective owners. FLEXlm and FLEXnet are trademarks of Flexera Software LLC.

Disclaimer Notice

THIS ANSYS SOFTWARE PRODUCT AND PROGRAM DOCUMENTATION INCLUDE TRADE SECRETS AND ARE CONFIDENTIAL AND PROPRIETARY PRODUCTS OF ANSYS, INC., ITS SUBSIDIARIES, OR LICENSORS. The software products and documentation are furnished by ANSYS, Inc., its subsidiaries, or affiliates under a software license agreement that contains provisions concerning non-disclosure, copying, length and nature of use, compliance with exporting laws, warranties, disclaimers, limitations of liability, and remedies, and other provisions. The software products and documentation may be used, disclosed, transferred, or copied only in accordance with the terms and conditions of that software license agreement.

ANSYS, Inc. and ANSYS Europe, Ltd. are UL registered ISO 9001: 2015 companies.

U.S. Government Rights

For U.S. Government users, except as specifically granted by the ANSYS, Inc. software license agreement, the use, duplication, or disclosure by the United States Government is subject to restrictions stated in the ANSYS, Inc. software license agreement and FAR 12.212 (for non-DOD licenses).

Third-Party Software

See the legal information in the product help files for the complete Legal Notice for ANSYS proprietary software and third-party software. If you are unable to access the Legal Notice, contact ANSYS, Inc.

Published in the U.S.A.

Table of Contents

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Introduction

Ansys ACT is a platform, providing a unified and consistent environment for quickly developing and deploying easy-to-use apps. With ACT, you can create apps to customize the following Ansys products:

- DesignModeler
- DesignXplorer
- Electronics Desktop
- Fluent
- Mechanical
- SpaceClaim
- Workbench

This guide offers an introduction to ACT, describing its capabilities and how to use them. While primarily intended for developers of ACT apps, it also provides information for end users who manage and execute apps.

Note:

Because ACT apps typically extend the functionality of Ansys products, ACT uses the terms *app* and *extension* interchangeably.

This first section describes the Ansys Product Improvement Program, provides an overview of how this guide is organized, supplies getting started information on using ACT, and provides important information related to licensing, migration, and known issues and limitations:

The Ansys Product Improvement Program

This product is covered by the Ansys Product Improvement Program, which enables Ansys, Inc., to collect and analyze *anonymous* usage data reported by our software without affecting your work or product performance. Analyzing product usage data helps us to understand customer usage trends

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

and patterns, interests, and quality or performance issues. The data enable us to develop or enhance product features that better address your needs.

How to Participate

The program is voluntary. To participate, select **Yes** when the Product Improvement Program dialog appears. Only then will collection of data for this product begin.

How the Program Works

After you agree to participate, the product collects anonymous usage data during each session. When you end the session, the collected data is sent to a secure server accessible only to authorized Ansys employees. After Ansys receives the data, various statistical measures such as distributions, counts, means, medians, modes, etc., are used to understand and analyze the data.

Data We Collect

The data we collect under the Ansys Product Improvement Program are limited. The types and amounts of collected data vary from product to product. Typically, the data fall into the categories listed here:

Hardware: Information about the hardware on which the product is running, such as the:

- brand and type of CPU
- number of processors available
- amount of memory available
- brand and type of graphics card

System: Configuration information about the system the product is running on, such as the:

- operating system and version
- country code
- time zone
- language used
- values of environment variables used by the product

Session: Characteristics of the session, such as the:

- interactive or batch setting
- time duration
- total CPU time used
- product license and license settings being used
- product version and build identifiers

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- command line options used
- number of processors used
- amount of memory used
- errors and warnings issued

Session Actions: Counts of certain user actions during a session, such as the number of:

- project saves
- restarts
- meshing, solving, postprocessing, etc., actions
- times the Help system is used
- times wizards are used
- toolbar selections

Model: Statistics of the model used in the simulation, such as the:

- number and types of entities used, such as nodes, elements, cells, surfaces, primitives, etc.
- number of material types, loading types, boundary conditions, species, etc.
- number and types of coordinate systems used
- system of units used
- dimensionality (1-D, 2-D, 3-D)

Analysis: Characteristics of the analysis, such as the:

- physics types used
- linear and nonlinear behaviors
- time and frequency domains (static, steady-state, transient, modal, harmonic, etc.)
- analysis options used

Solution: Characteristics of the solution performed, including:

- the choice of solvers and solver options
- the solution controls used, such as convergence criteria, precision settings, and tuning options
- solver statistics such as the number of equations, number of load steps, number of design points, etc.

Specialty: Special options or features used, such as:

- user-provided plug-ins and routines

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- coupling of analyses with other Ansys products

Data We Do Not Collect

The Product Improvement Program does *not* collect any information that can identify you personally, your company, or your intellectual property. This includes, but is not limited to:

- names, addresses, or usernames
- file names, part names, or other user-supplied labels
- geometry- or design-specific inputs, such as coordinate values or locations, thicknesses, or other dimensional values
- actual values of material properties, loadings, or any other real-valued user-supplied data

In addition to collecting only anonymous data, we make no record of where we collect data from. We therefore cannot associate collected data with any specific customer, company, or location.

Opting Out of the Program

You may *stop* your participation in the program any time you wish. To do so, select **Ansys Product Improvement Program** from the Help menu. A dialog appears and asks if you want to continue participating in the program. Select **No** and then click **OK**. Data will no longer be collected or sent.

The Ansys, Inc., Privacy Policy

All Ansys products are covered by the Ansys, Inc., Privacy Policy.

Frequently Asked Questions

1. *Am I required to participate in this program?*

   No, your participation is voluntary. We encourage you to participate, however, as it helps us create products that will better meet your future needs.

2. *Am I automatically enrolled in this program?*

   No. You are not enrolled unless you explicitly agree to participate.

3. *Does participating in this program put my intellectual property at risk of being collected or discovered by Ansys?*

   No. We do not collect any project-specific, company-specific, or model-specific information.

4. *Can I stop participating even after I agree to participate?*

   Yes, you can stop participating at any time. To do so, select **Ansys Product Improvement Program** from the Help menu. A dialog appears and asks if you want to continue participating in the program. Select **No** and then click **OK**. Data will no longer be collected or sent.

5. *Will participation in the program slow the performance of the product?*

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

No, the data collection does not affect the product performance in any significant way. The amount of data collected is very small.

6. *How frequently is data collected and sent to Ansys servers?*

   The data is collected during each use session of the product. The collected data is sent to a secure server once per session, when you exit the product.

7. *Is this program available in all Ansys products?*

   Not at this time, although we are adding it to more of our products at each release. The program is available in a product only if this *Ansys Product Improvement Program* description appears in the product documentation, as it does here for this product.

8. *If I enroll in the program for this product, am I automatically enrolled in the program for the other Ansys products I use on the same machine?*

   Yes. Your enrollment choice applies to all Ansys products you use on the same machine. Similarly, if you end your enrollment in the program for one product, you end your enrollment for all Ansys products on that machine.

9. *How is enrollment in the Product Improvement Program determined if I use Ansys products in a cluster?*

   In a cluster configuration, the Product Improvement Program enrollment is determined by the host machine setting.

10. *Can I easily opt out of the Product Improvement Program for all clients in my network installation?*

    Yes. Perform the following steps on the file server:

    a. Navigate to the installation directory: `[Drive:]\v222\commonfiles\globalsettings`

    b. Open the file **ANSYSProductImprovementProgram.txt**.

    c. Change the value from "on" to "off" and save the file.

Guide Overview

This section introduces ACT. Subsequent sections organize content as follows:

- ACT Overview (p. 19): Presents a high-level overview of ACT that introduces you to extension development, customization capabilities, documentation and development resources, and "roadmaps" for specific types of customizations.
- ACT Tools (p. 39): Describes the many tools that ACT provides for developing, debugging, and executing extensions.
- Extensions (p. 71): Describes the basic components of an ACT scripted extension and provides information about extension configuration, supplied examples, installed IronPython function libraries, advanced programming, and extension creation.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Feature Creation (p. 127): Provides examples of customization capabilities common to the Ansys products that support using ACT to leverage existing product functionality and add custom functionality and operations.
- Simulation Wizards (p. 141): Describes wizards and how you can create them to walk non-expert users step-by-step through simulations.
- Debugging (p. 169): Describes debug mode and the methods for debugging scripts during extension development.
- ACT Customization Guides for Supported Ansys Products (p. 183): Provides links to additional guides that explain how to use ACT to customize specific Ansys products.

Getting Started with ACT

Ansys offers class-leading, off-the-shelf simulation technologies. To most effectively deploy pervasive simulation, you may desire a more curated experience to match our simulation expertise with your user, company, or industry needs. Ansys ACT equips you with the power to customize and extend the Ansys experience.

This section helps you to get started with ACT by answering these questions:

What is customization?

What is extensibility?

What is ACT?

What capabilities does ACT provide?

What skills are required for using ACT?

How do I begin using ACT?

Where can I find published ACT apps?

What is customization?

Customization is the in-product operation of modifying existing functionality and exposing brand new features. The goal of customization is to alter an existing user experience to match the needs of the user.

What is extensibility?

Building on customization, extensibility operates at a higher level of custom user experience, out-of-product feature exposure, and vertical applications. The goal of extensibility is to reliably and consistently enhance a software package with minimal development and maintenance effort. To extend Ansys software, you create extensions that add and modify functionality while minimizing the impact on existing functionality.

What is ACT?

ACT is a unified and consistent toolkit for customizing and extending Ansys products, providing the fastest and easiest way to create simulation engineering apps that meet your specific needs. While tailoring simulation applications to fit your use has traditionally been complex and time-consuming,

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

ACT simplifies this process, allowing you to focus more on simulation analysis than on customizing software.

ACT uses easy-to-learn yet powerful eXtensible Markup Language (XML) and IronPython programming languages. Even if you are a non-expert user, you can create custom apps for your advanced workflows. Unlike typical software programming, ACT does not require a commercial Integrated Development Environment (IDE). Instead, it provides an intuitive development environment with tools, documentation, and lots of examples to guide you through the process. With ACT, you can create customizations in only hours or days instead of weeks or months.

Many Ansys products expose their own scripting solutions. However, ACT provides a single scripting solution for customizing all Ansys products. You can mix ACT APIs (Application Programming Interfaces) with product-specific APIs without needing to compile external code or link with existing Ansys libraries. Additionally, ACT enables you to manage the interfaces between these products and the additional customizations, ensuring that they all interact accurately.

ACT's intuitive APIs and simple app creation tools capture the best practices of expert engineering analysts, thereby reducing training and implementation costs and enabling a broader range of engineers and designers to effectively use simulation tools. By fostering a unified simulation workflow, ACT allows you to integrate your non-Ansys engineering tools and data into the Ansys ecosystem to maximize the productivity of your engineering teams. Leverage these streamlined simulation workflows for faster and better decisions throughout the entire product life cycle, from concept to product use.

What capabilities does ACT provide?

While Ansys provides comprehensive engineering simulation solutions across all physics areas, you are likely to have business-specific needs that are not native to Ansys products. To tailor Ansys products to meet these needs, ACT provides customization capabilities, categorized into three types:

Feature Creation

Simulation Workflow Integration

Process Compression

Note:

For additional information about ACT, including "roadmaps" designed to guide you through the steps of creating the desired type of ACT customization for an Ansys product, see ACT Overview (p. 19).

Feature Creation

Feature creation is the direct, API-driven *customization* of Ansys products. In addition to leveraging the functionality already available in a product, ACT enables you to add functionality and operations of your own. Your custom additions operate as "native" features in the target Ansys product.

Feature creation (p. 127) examples include:

- Creating custom loads and geometries
- Adding custom preprocessing and postprocessing features

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Integrating third-party solvers, sampling methods, and optimization algorithms

These Ansys products support feature creation:

- DesignModeler
- DesignXplorer
- Fluent
- Mechanical
- Workbench

Feature creation capabilities common to all these products include user interface entry creation, ACT-based property creation, and property parameterization. Product-specific feature creation capabilities are described in the ACT customization guide for the product. The concluding (p. 183) section provides links to these guides.

A few examples follow of feature creation in various Ansys products:

DesignModeler

Define new user interface elements to execute custom actions

Expose new geometry features

Fluent

Deploy your own UDFs with ACT UDF macro encapsulation

Mechanical

Integrate your legacy APDL macros with APDL encapsulation

Develop your own criterion using IronPython and integrate it into the powerful Mechanical Post environment

Add new boundary conditions, preprocessing and postprocessing algorithms, and custom results

Define new user interface elements to execute custom actions

Workbench

Define new user interface elements to execute custom actions

Feature creation also includes the integration of custom solvers and sampling and optimization algorithms. For example, you can:

- Integrate new sampling and optimization methods in DesignXplorer. ACT enables you to extend the existing set of Ansys-supplied sampling and optimization algorithms with your own in-house, third-party, or commercial strategies. You can then select these integrated methods from DesignXplorer's setup properties.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Integrate custom solvers directly in Mechanical. By embedding your solvers into Mechanical simulations, you can create new Workbench analysis systems based on your technologies. Within Mechanical, the solver operates seamlessly with the processing routines for the native mathematical model, solver, and results processing.

Simulation Workflow Integration

Simulation workflow integration is the incorporation of external knowledge such as apps, processes, and scripts into the Ansys ecosystem. With ACT, you can create custom simulation workflows that can be inserted on the flowchart-like schematic in Workbench.

An engineering simulation workflow is a sequence of actions performed on well-defined data to obtain insightful results. A typical simulation workflow can be broken down into five steps:

1. Define or retrieve input data
2. Prepare for execution
3. Run the process
4. Generate output data
5. Publish results

With ACT, you define each workflow step as a custom task (component) and then combine multiple custom tasks into a custom task group (system). You can then insert custom task groups on the Workbench **Project Schematic** to construct consistent and cohesive simulation workflows, allowing your business-specific elements to coexist and interface with pre-built Ansys solutions. Currently, Workbench is the only Ansys product to support simulation workflow integration. For more information, see the *ACT Customization Guide for Workbench*.

You can use integrated workflows for various simulation scenarios, performing custom behaviors such as specialized data processing, report generation, and application synchronization to manage and integrate heterogeneous processes and tools into your Ansys simulation environment. By using ACT to integrate diverse data generated from in-house and commercial engineering software products into your Ansys environment, you can improve your product lines and innovate faster.

Examples follow of using custom workflows to integrate external data and applications:

- Maintain or reuse data from an external application, such as a CAD system. With ACT, you can automate the process of mapping and linking to this data to save time and reduce errors, greatly improving productivity. Additionally, you can use ACT to batch process the generation of images and reports from your model results.
- Expose external applications and data with custom workflows in Workbench to achieve:
  - Native "OEM" look and feel
  - Professional-grade application deployment
  - Project data and file management solutions
  - Collaboration with industry-leading applications

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Flexible project construction
- Remote Solve Manager (RSM) job submission to send large jobs to more powerful remote high-performance machines

Process Compression

Process compression is the encapsulation and automation of processes available in one or more Ansys products. The result is a simulation wizard (p. 141), which incorporates your best practices to reliably guide non-expert users step-by-step through a complex simulation to produce desired results quickly. This type of ACT customization provides the greatest breadth because wizards simplify complex processes, allowing them to be repeated whenever needed. You can create wizards for all products listed in the introduction (p. 1).

With wizards, you can manipulate existing features and simulation components, organizing them as needed to produce a custom automated process. A wizard can compress and automate processes within a single Ansys product to manage complex model interactions for one physics area. Or, a wizard can compress and automate processes of multiple Ansys products and even external applications for complex multi-physics simulations. By using wizards to simplify the analytical process, you boost the performance of the entire enterprise.

Wizards allow you to leverage both the existing functionality of Ansys products and the scripting capabilities of the Workbench framework API. For example, you can copy a journal of the steps that you take in an Ansys product into an ACT script and then parametrize the syntax to make the script generic for reuse.

The degree of automation possible depends on the product being customized. Examples of a Workbench-based project wizard and a mixed wizard appear in this guide. Examples of wizards specific to an Ansys product are described in the ACT customization guide for the product. The concluding (p. 183) section provides links to these guides.

Because simulation workflows are often so complex, the use of simulation tends to be limited to only a few expert users, reducing the overall efficiency of the engineering process. The need to satisfy process compliance requirements and ensure data integrity adds even more complexity. Using wizards, you can leverage the knowledge of your engineering process experts by compressing your business-specific processes into streamlined, simple-to-follow steps. Because each step exposes only the critical information required to ensure successful simulation, wizards shield users from technical complexities.

What skills are required for using ACT?

The development of ACT apps requires some knowledge of XML and IronPython. An ACT app begins as a scripted extension (p. 71) consisting of an XML file that defines and configures the content of the extension and at least one IronPython script that defines the functions invoked by user interactions, implementing the extension's behavior.

With ACT's standard XML and IronPython approach for all Ansys products, app creation is easy to learn, regardless of the Ansys products that you plan to customize. Apply the same consistent framework and APIs to meet your customization objectives and workflows.

The ACT App Builder (p. 94) makes creating and editing XML files and IronPython scripts even easier. Rather than manually performing these app-building actions, you can use this tool to automatically

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

generate reusable scripts for customization in an interactive environment, without having to write code. Built-in journal recording eliminates manual callback programming and property substitutions, dramatically simplifying and speeding up customization. Intuitive property definition and processing specific to supported Ansys products breaks down barriers to scripting. Use of the **ACT App Builder** ensures easier, faster, and cheaper development of ACT apps.

For apps that customize Ansys solvers, knowledge of APDL is required. Additionally, advanced users can take advantage of the opportunities of .NET integration and support provided by IronPython. If you are interested in specific members or code methods, the ACT Console (p. 53) is a convenient tool to interactively navigate the APIs through robust autocompletion.

How do I begin using ACT?

The ACT Start Page (p. 39) provides a central place from which to access the many tools (p. 39) that ACT provides for developing, debugging, and executing extensions (p. 71). From here, you can access the ACT Console (p. 53), which exposes the ACT APIs. As a novice, you can discover APIs and write small and simple scripts for automating routine tasks that you perform within an Ansys product. As your experience and confidence grows, you can write full-fledged extensions that customize and extend Ansys products.

The Extension Manager (p. 41) provides for installing and loading extensions. The Wizards launcher (p. 47) starts simulation wizards (p. 141), and the binary extension builder (p. 49) creates compiled binary files from scripted extensions. With tools like the ACT App Builder (p. 94), ACT Debugger (p. 170), and ACT Workflow Designer, modifying existing Ansys functionalities and adding new custom functionalities and workflows is easy, giving you the power to decide how Ansys products should look and behave.

ACT also provides a full set of supporting resources to help you with app development. The ACT documentation includes comprehensive feature overviews and detailed API descriptions. To aid you in development efforts, ACT supplies comprehensive examples (p. 81) for various types of Ansys product customizations, all of which are designed to help you to understand how to use ACT to develop apps. For instance, you can easily modify supplied examples to align them with your own simulation vision, saving you both development time and money. Written and tested on supported Windows and Linux platforms, these examples are packaged for download from the help panel on the ACT Start Page (p. 39). Additionally, on the App Developer Resources page, the **Help & Support** tab displays a link for downloading the examples.

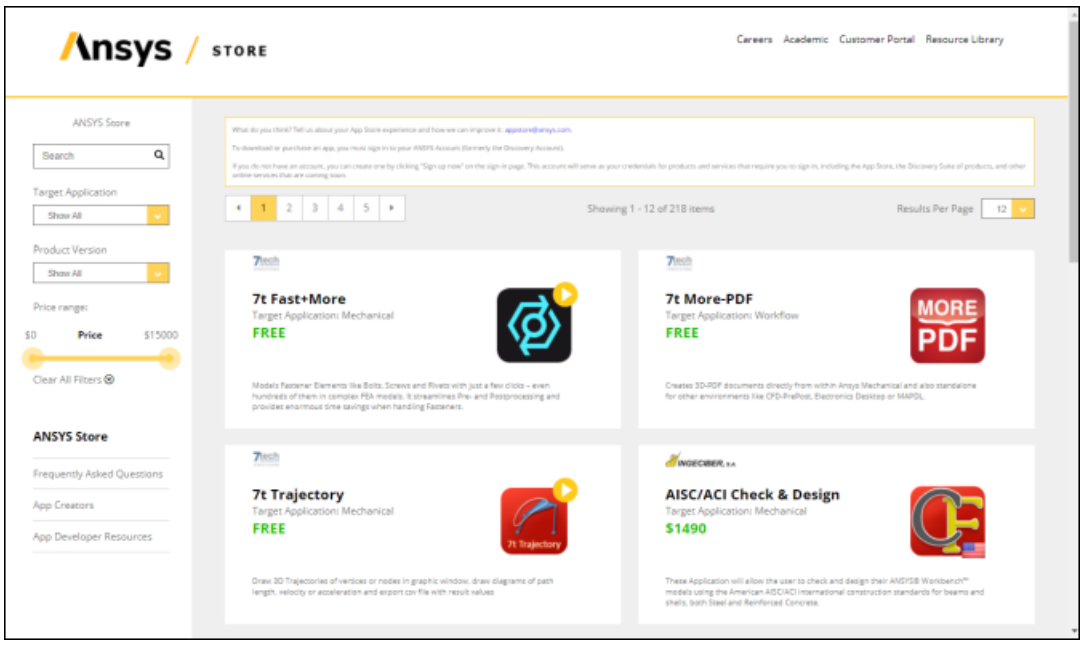
The Ansys training course for ACT covers using ACT and its APIs to customize Workbench, Mechanical/Meshing, and DesignModeler. After completing this course, you should be able to automate the creation of standard tree objects in Mechanical/Meshing, create custom loads and results in Mechanical, create custom objects in DesignModeler, and create wizards in Workbench or its integrated modules.

Where can I find published ACT apps?

The Ansys Store provides hundreds of free and paid apps developed by Ansys and trusted partners. With more than 6,000 downloads per month, the Ansys Store provides an ever-expanding library of dynamic simulation solutions. These business-specific Ansys apps range greatly in functionality and complexity. You can filter apps based on target application, product version, and price range. Because free apps are available as simple downloads, you can easily use the Extension Manager (p. 41) to

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

install and load them to better explore the power that ACT provides. For some free apps, the source code is also included so that you can view and even modify it to create your own custom app.



In the left pane, you can click the link for the App Developer Resources page, where you will find a multitude of resources for using ACT to develop your own custom apps. On this page, the **Post My ACT Application** tab displays a link for downloading instructions for posting your custom apps on the Ansys Store.

By making complex simulations accessible to all engineers, Ansys apps "democratize" the use of simulation across your organization. With designers being able to perform simulations, expert engineers can spend more time on researching and developing innovative and reliable products.

Licensing

The ACT license is bundled with most Ansys 2022 R2 products. You must have a license to develop scripted extensions. You do not need to have a license to run binary extensions (WBEX files).

The ACT license enables two main capabilities:

- Ability to load scripted extensions for development, debugging, and execution. When an ACT license is checked out, scripted extensions can be loaded. No matter how many scripted extensions are loaded, only one ACT license is checked out. This license is released once all scripted extensions are unloaded. If no ACT license is checked out, only binary extensions can be loaded.
- Ability to build binary extensions from scripted extensions. The ACT license is checked out when a build starts and is released once the build finishes. If an ACT license is already checked out by a loaded scripted extension, the building of a binary extension does not require a second ACT license.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Migration Notes

As improvements are made to ACT APIs and the way that they display and transmit data, great efforts are taken to ensure that changes are backwards-compatible. In this release, no API changes were made that might impact your existing ACT extensions. However, for migration notes specific to scripting in Ansys Mechanical, see Mechanical API Migration Notes in the *Scripting in Mechanical Guide*.

Note:

- On the Ansys Store, the App Developer Resources page displays a link on the **Help & Support** tab to the stand-alone *ACT Migration Notes* document. This document may contain entries that were added after publication of this guide. To quickly access this document, click here.
- ACT has superseded the Ansys Workbench Software Development Kit (SDK) and External Connection Add-In as the best-in-class tool set for customizing Ansys products. Support for the SDK and External Connection Add-in ended as of 19.0. If you have used these deprecated tool sets for Workbench customizations, click here to quickly access the migration guide. On the App Developer Resources page, the **Help & Support** tab also displays a link to this migration guide.

Known Issues and Limitations

This section lists known issues and limitations in the ACT 2022 R2 release.

Note:

- For known issues and limitations specific to Mechanical APIs, see Mechanical API Known Issues and Limitations in the *Scripting in Mechanical Guide*.
- On the Ansys Store, the App Developer Resources page displays a link on the **Help & Support** tab to the stand-alone *ACT Known Issues and Limitation* document. This document might contain entries that were added after publication of this guide. To quickly access this document, click here.

Captions for custom objects cannot end in a space followed by a number

When naming a caption for a custom object, the software automatically appends a number to the name if it is identical to existing objects. Do not end the name with a space followed by a number. While no error appears, the number will be removed and may be changed. **Workaround**: When renaming a caption for a custom object, always follow a space with a non-numeric character before using a number. For example, rather than renaming a caption to **`my custom object 2`**, rename it to **`my custom object n2`**.

Third-party solver fails to display in Workbench

A third-party solver exposed through the **`<solver>`** simdata tag may not display in the Workbench **Toolbox** if the XML file specified for the attribute **`system_template`** relies on a

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

`ValidationCode`. To obtain a correct **`ValidationCode`** value compatible with recent licensing system upgrades, contact your local support channel.

ACT panels may not fill entire dialog width

Some ACT panels, such as the **ACT Console** and **ACT Debugger**, may not "spring" the entire width of the parent dialog. Panel behavior and user interaction are unaffected.

Reloading a Workbench extension while in use

A Workbench extension can define new systems to insert in the **Project Schematic**. Reloading such an extension after creating new system instances will result in unexpected behavior. Before reloading, delete all extension-related systems or save and reset the project.

In HTML components, the URL protocol must be used for image, custom JS, and CSS files

In an HTML component, an image, custom JS, or CSS file does not display unless the file URL protocol (**`file:///`**) is placed before the file path.

Prior to 2020 R2, an absolute path could be used:

```
img src="c:/Users/User1/myimage.jpg"
```

However, the **`file:///`** protocol must now be placed before the file path. Some examples follow.

For an image file:

```
img src="file:///c:/Users/User1/myimage.jpg"
```

For a custom JS file:

```
component.CustomJSFile = r'file:///' + str(ExtAPI/Extension.InstallDir)+ r'\custom_table.js'
```

After installation, the ACT Start Page sometimes appears blank

If after installing Ansys products, you see that the **ACT Start Page** is blank, right-click inside the blank window and select **Reload**.

VPN conflicts can arise if IP information is retained

When using a VPN, IP information can be retained, even after turning off the VPN and rebooting the computer. The workaround is to release the IP configuration using the following script:

```
echo Releasing IP
ipconfig /release > NUL
echo Acquiring new IP
ipconfig /renew > NUL
```

Enabled FIPS security policy

When running with an enabled FIPS security policy, ACT User Interface elements are not supported.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Workbench customization

When using ACT to create Workbench custom workflows, not all optional attributes for the tag <property> are currently supported. For example, **`visibleon`** is not supported.

For DesignModeler, the ACT Console, Extensions Log, and ACT wizards are supported on Windows platform only

For DesignModeler, the **ACT Console**, **Extensions Log**, and ACT wizards are supported only when DesignModeler is running on the Windows platform.

ACT for Fluent is supported on Windows platform only

When using ACT for Fluent customization, ACT is supported only when Fluent is running on the Windows platform.

For Fluent, the initial attempt at opening the Extensions Log from the ACT Start Page causes an unexpected shutdown

In a new Fluent installation, the initial attempt at opening the **Extensions Log** from the **ACT Start Page** does not display the log dialog box. Instead, Fluent shuts down unexpectedly. However, subsequent attempts at opening this dialog box are successful.

ACT Debugger and App Builder are supported on Windows platform only

The **ACT Debugger** is supported only on the Windows platform from the **Project** page in Ansys Workbench and from DesignModeler and Mechanical. The **ACT App Builder** is supported only on the Windows platform.

ACT App Builder is not supported in Ansys Electronics Desktop

The **ACT App Builder** is not supported in Ansys Electronics Desktop (AEDT). Clicking the **Open App Builder** icon on the **ACT Start Page** in AEDT will not start the App Builder.

Limited ACT localization support

Localization of ACT is limited to the languages currently supported in Ansys Workbench. This limitation does not apply to the ability to manage various languages within the extension. For example, the property names created by an extension do not have to be in the same language as the current activated language in Workbench.

There is no mechanism to integrate localization for the property names defined by an extension. To manage different languages for your property names, you must develop localization yourself. Both regional settings based on the "." or the "," decimal symbol are available. However, the implementation of the extension should use the "." symbol for any value defined at the XML or IronPython level.

Default location of ACT extensions for SpaceClaim

The default location in which SpaceClaim is to look for ACT extensions is `%ANSYSversion_DIR%\scdm\Addins`. However, SpaceClaim is not currently recognizing this location. Workarounds include either installing extensions in `%APPDATA%\Ansys\v222\ACT\extensions` or using the gear icon on the graphic-based **Ex-**

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

tension Manager accessed from the **ACT Start Page** to add the folder for the default location or the folder to which you installed the extension.

Mechanical read-only mode

ACT does not support Mechanical read-only mode. If you launch Mechanical and see read-only mode, avoid interactions with ACT extensions and features as changes could result in unexpected application behavior.

Captions for custom objects cannot contain a space followed by a number

When renaming a caption for a custom object, do not follow a space with a number. While no error appears, the caption is not renamed. **Workaround**: When renaming a caption for a custom object, always follow a space with a non-numeric character before using a number. For example, rather than renaming a caption to "my custom object 2", rename it to "my custom object n2".

ExtAPI.Extension.Attributes does not support OrderedDict values

When **`ExtAPI.Extension.Attributes`** is used to store a value of type **`OrderedDict`**, an empty dictionary is stored instead of the supplied value.

Saving dictionaries in attributes slows the opening of Mechanical significantly

When loading a project with custom ACT objects whose attributes have been filled with dictionaries, it takes Mechanical much more time to open. In cases where these dictionaries are very large, Mechanical may be unable to recreate results as ACT results, displaying user-defined results instead.

Accessing UserInterface on ExtAPI in Mechanical from an onInit callback or globally causes errors

Accessing **`UserInterface`** on **`ExtAPI`** in Mechanical from an **`onInit`** callback or globally causes errors to be thrown in the **Extensions Log**, resulting in instability. **Workaround:** Access **`UserInterface`** from an **`onLoad`** callback or any place other than the **`onInit`** callback. Keep in mind that an **`onLoad`** callback is invoked only when Mechanical is started. Reloading an extension using the button in the GUI will not invoke an **`onLoad`** callback.

Electronics Desktop module CoreGlobalScriptContextFunctions is unavailable

The Electronics Desktop module **`CoreGlobalScriptContextFunctions`** is unavailable in ACT. The following functions are available only through the **Command** window in Electronics Desktop:

- **`AddErrorMessage`**
- **`AddFatalMessage`**
- **`AddInfoMessage`**
- **`AddWarningMessage`**
- **`LogDebug`**
- **`LogError`**

As alternatives, you can use the following:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- **`oDesktop.AddMessage`**
- Python logging functionality (for example, **`"import logging"`** and use the logging module)

Limitations Unique to Linux

If you are running on Linux, you should be aware of the following limitations:

Starting the ACT Start Page can cause an unexpected shutdown

When you start the **ACT Start Page** on Linux, an unexpected shutdown can occur. On some Linux operating system variants such as Red Hat, removing the package **`totem-mozplugin`** resolves the issue:

```
yum remove totem-mozplugin
```

Some Linux versions contain a JPEG library that conflicts with the Ansys package

In some Linux versions, a JPEG library conflicts with the Ansys package. When using ACT wizards with JPEG images, this conflict can produce an SIGSEV error and an application crash. If this occurs, you can convert your images to other file formats such as GIF or PNG.

Graphics API issues in Ansys DesignModeler and Mechanical when no extensions are loaded

When no extensions are loaded, there are some limitations on the **`Graphics`** API from the **ACT Console** in Ansys DesignModeler and Mechanical. For instance, Factory2D does not work. Therefore, you should load one or more extensions before using the **`Graphics`** API from the **ACT Console**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.

ACT Overview

ACT is a unified and consistent toolkit for the customization and expansion of Ansys products. Using ACT, you can create extensions to tailor the Ansys products listed in the introduction (p. 1) to meet your product-specific and multi-physics simulation needs. You can also engage the Ansys simulation ecosystem through integration and custom workflow techniques.

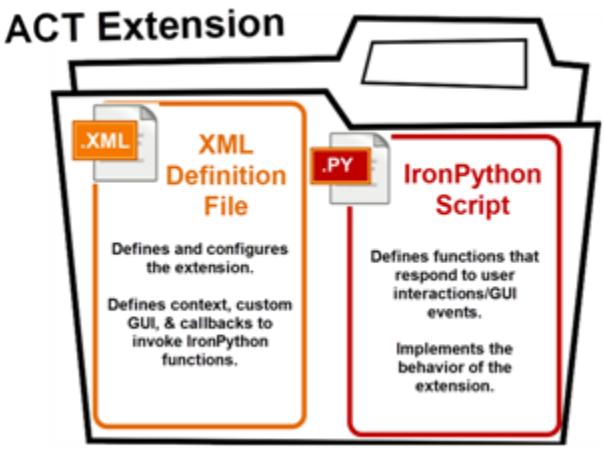
The purpose of this document is to:

- Present a high-level overview of ACT, introducing you to the general concepts that you need to know to develop extensions.
- Help you to start developing your own extensions, from the initial design phase through the implementation and—when desired—to the final posting of your app to the Ansys Store. On the App Developer Resources page of the Ansys Store, the **Post My ACT Application** tab displays a link for downloading posting instructions.
- Provide references and links to ACT documentation and resources so that you can easily access relevant information, such as:
  - Sections in ACT guides that specifically address your objective
  - Reference guides containing API and XML information
  - ACT extension and template examples
- Deliver specialized customization "roadmaps" designed to guide you through the steps of creating the desired type of ACT customization for an Ansys product.

Extension Structure

All ACT extensions are made up of the same two basic parts: an XML extension definition file and an IronPython script.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Extensions can potentially include additional components such as external Python libraries, C# code, input files, and images to display in the custom interface. For comprehensive information, see Extensions (p. 71).

Extension Formats

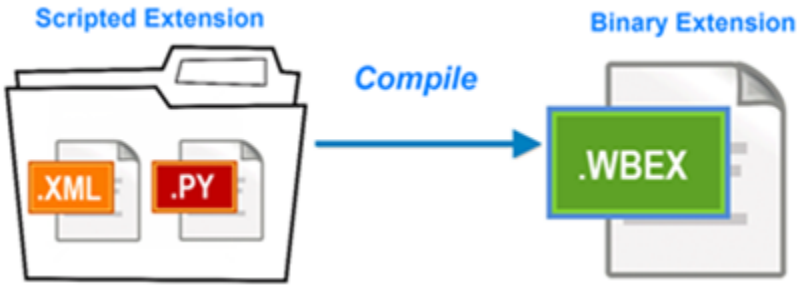
ACT extensions can be created in two different formats: scripted and binary.

Scripted Extensions

A scripted extension consists of an editable XML extension definition file and IronPython script. The XML file and the folder containing the IronPython script and any supplementary supporting files are saved at the same level. For more information, see Setting Up the Directory Structure (p. 74)

Binary Extensions

A binary extension is a compiled WBEX file resulting from the build of a scripted extension. Binary extensions can be shared with others, who are able to execute the extension but cannot edit it. For more information, see Binary Extension Builder (p. 49).



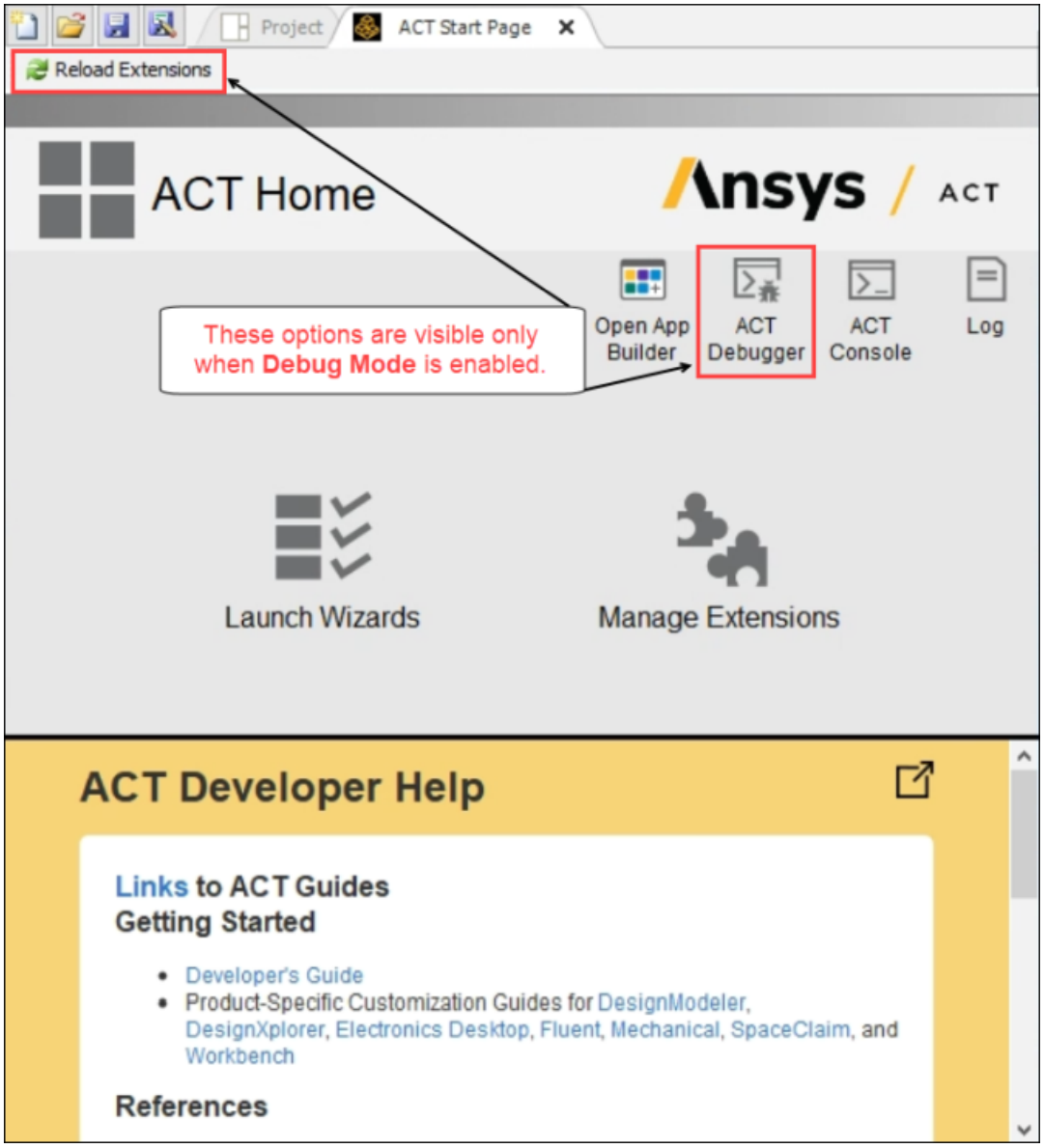
Note:

You need an ACT license to load scripted extensions and to build binary extensions. For more information, see Licensing (p. 12).

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

ACT Customization Tools

You can access most of ACT's customization tools (p. 39) from the ACT Start Page (p. 39), which is available in all Ansys products that support extensions. The way that you access this page depends on the Ansys product that you are using.



ACT Customization Capabilities

In the introduction, What capabilities does ACT provide? (p. 7) describes the three types of ACT customizations:

- Feature creation
- Simulation workflow integration

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Process compression 

In subsequent roadmaps, color coding is assigned to distinguish between customization types. This table shows the Ansys products that currently support ACT customization capabilities.

Product	**Feature Creation** 	**Simulation Workflow Integration** 	**Process Compression** 
DesignModeler	✔		✔
DesignXplorer	✔		✔
Electronics Desktop			✔
Fluent	✔		✔
Mechanical	✔		✔
SpaceClaim			✔
Workbench	✔	✔	✔

ACT Documentation and Resources

The following topics introduce you to the ACT documentation and resources that are available to help you use ACT:

ACT General Documentation

ACT Reference Documentation

ACT Templates, Training, and More

Visit the Ansys Store to view the apps offered by Ansys and its customers. You can filter posted apps based on product family, version, and by free or paid apps.

ACT General Documentation

This guide introduces you to ACT and concludes (p. 183) with links to product-specific ACT customization guides. The help panel for the ACT Start Page (p. 39) provides links to these guides and other ACT documents.

Additionally, the App Developer Resources page displays tabs with links to these and other ACT documents. Documents of possible interest include stand-alone migration notes, a presentation providing instructions for installing ACT extensions, and the software licensing agreement and instructions for publishing extensions to the Ansys Store.

ACT Reference Documentation

ACT supplies the following reference documentation:

- ACT API Reference Guide: Provides interface-level API information for ACT.
- ACT XML Reference Guide: Provides XML element definition and tagging information for ACT.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Workbench Scripting Guide: Provides a general introduction to journaling and scripting capabilities that can be used in some customizations. Also contains product-specific data container information.
- Ansys ACT API and XML Online Reference Guide: Provides an easy-to-navigate HTML document in which you can quickly look up all three levels of API information (global, automation, and object), along with XML element definition and tagging information. This online version is the recommended reference document for ACT developers. If you want, you can download a ZIP file containing this document. Additionally, the help panel for the ACT Start Page (p. 39) provides links for both the online and downloadable versions.

ACT Templates, Training, and More

ACT supplies additional support materials:

Extension and Template Packages

Extension and template packages (ZIP files) provide examples for learning how to use ACT. You can download these packages from the help panel for the ACT Start Page (p. 39). Additionally, on the Ansys Store, the App Developer Resources page provides links for downloading extension examples from the **Help & Support** tab and templates from the **ACT Templates** tab. For package descriptions and download information, see Extension and Template Examples (p. 81).

Training Materials

Training materials for ACT and IronPython are available on the Ansys Learning Hub.

Case Studies & Presentations

On the App Developer Resources page, the **Case Studies & Presentations** tab provides links for downloading ACT case studies and presentations. Additionally, the ACT product page provides links to other case studies and product webinar information.

Roadmaps for ACT Customization

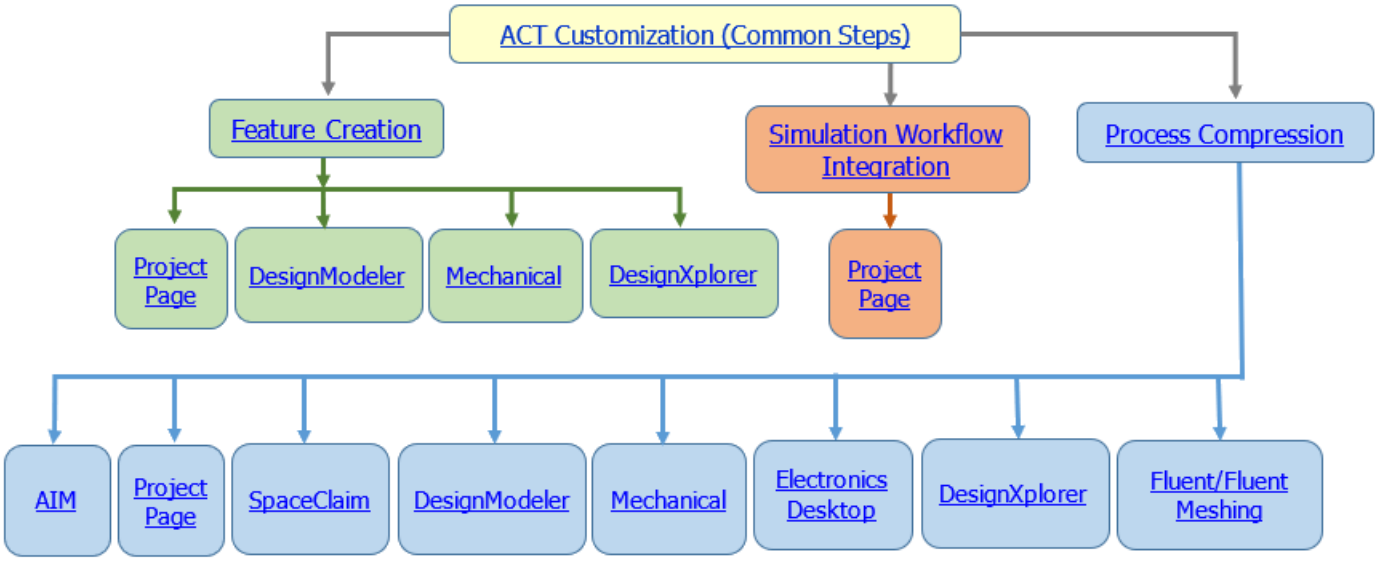
Subsequent sections contain development roadmaps to guide you through the extension creation process. Each roadmap provides a high-level description of the steps necessary for a specific customization. It then directs you to more detailed information in ACT guides.

Note:

Unless otherwise indicated, references are to topics in this guide.

Content goes from general to specific. The following image displays the overall roadmap for ACT customization.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



The next section describes the steps that are common to all extensions. Subsequent sections provide roadmaps and steps for each of the three customization types. Each section for a customization type provides information for the specific Ansys products that can be customized.

ACT Customization – Common Steps

The following tables provide the steps that are common to all extensions. References are to topics in the ACT Developer's Guide (p. 1).

Step 1: Define Your Extension		
Step	**Description**	**Reference**
1a	Create the XML definition file.	Extensions (p. 71) > Extension Definition (p. 72) > Creating the Extension Definition File (p. 72)
1b	Create the IronPython script.	Extensions (p. 71) > Extension Definition (p. 72) > Creating the IronPython Script (p. 73)
1c	Place the files in the correct directory structure.	Extensions (p. 71) > Extension Definition (p. 72) > Setting Up the Directory Structure (p. 74)
1d	Optional: Compile the scripted extension into a binary extension (WBEX file).	ACT Tools (p. 39) > Binary Extension Builder (p. 49)
1e	Configure extension options.	Extensions (p. 71) > Extension Configuration (p. 78)

Step 2: Install and Load Your Extension		
Step	**Description**	**Reference**
2a	Install your extension.	ACT Tools (p. 39) > Extension Manager (p. 41) > • Installing and Uninstalling Scripted Extensions (p. 44)

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Step 2: Install and Load Your Extension		
Step	**Description**	**Reference**
		• Installing and Uninstalling Binary Extensions (p. 45)
2b	Load your extension.	ACT Tools (p. 39) > Extension Manager (p. 41) > • Loading and Unloading Extensions (p. 46) • Configuring Extensions to Load by Default (p. 46)

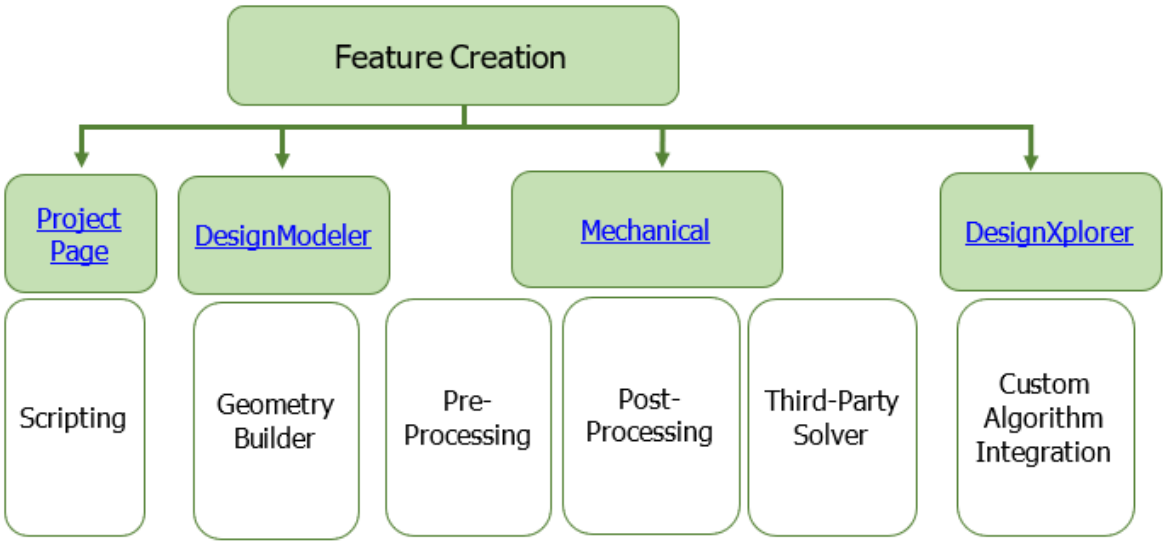
Step 3: Debug Your Extension		
Step	**Description**	**Reference**
3	Optional: Debug your extension using ACT tools or external tools.	Debugging (p. 169) > • Debug Mode (p. 169) • ACT Debugger (p. 170) • Debugging with Microsoft® Visual Studio (p. 181)

Feature Creation Roadmap

Feature creation refers to using the ACT API to create custom features for a product. The following roadmap shows:

- Products for which ACT feature creation capabilities are available
- Types of features that can be created for each product

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Note:

Fluent does not currently support using ACT to deliver custom features via extensions. However, it does support using UDF libraries for feature creation. For more information, see UDF Folder in the *ACT Customization Guide for Fluent*.

All products that support ACT-based feature creation provide the following capabilities:

Topic	Reference
Common capabilities	Feature Creation (p. 127) > • Toolbar Creation (p. 127) • Dialog Box Creation (p. 132) • Extension Data Storage (p. 132) • ACT-Based Property Creation (p. 134) • ACT-Based Property Parameterization (p. 138)

The following topics describe feature creation capabilities in specific Ansys products:

Feature Creation in DesignModeler

Feature Creation in DesignXplorer

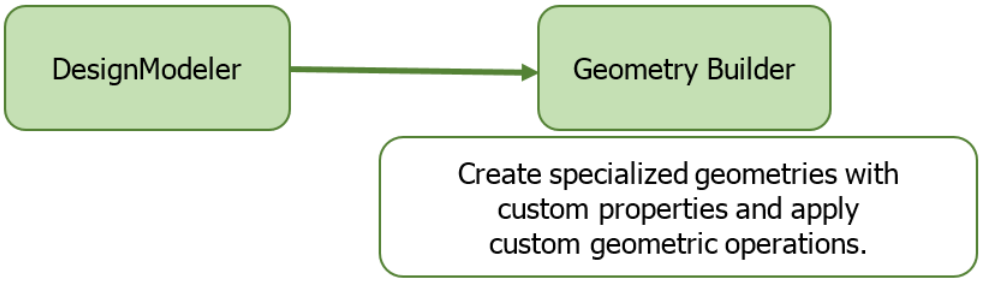
Feature Creation in Mechanical

Feature Creation in the Workbench Project Page

Feature Creation in DesignModeler

In DesignModeler, you can create custom **Geometry Builder** features.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



In addition to the common capabilities listed in Feature Creation Roadmap (p. 25), DesignModeler supports the feature creation capabilities that follow. References in this table take you to topics in the *ACT Customization Guide for DesignModeler*.

Topic	Reference
Capabilities specific to DesignModeler	DesignModeler Feature Creation > • ACT-Based Property Parametrization in DesignModeler • ACT-Based Geometry Creation
DesignModeler – API usage examples	DesignModeler APIs > • Using the Selection Manager in DesignModeler • Creating Primitives • Applying Operations

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

In the API information, see these DesignModeler namespaces:

- **`Ansys.ACT.Interfaces.DesignModeler`**
- **`Ansys.ACT.Automation.DesignModeler`**

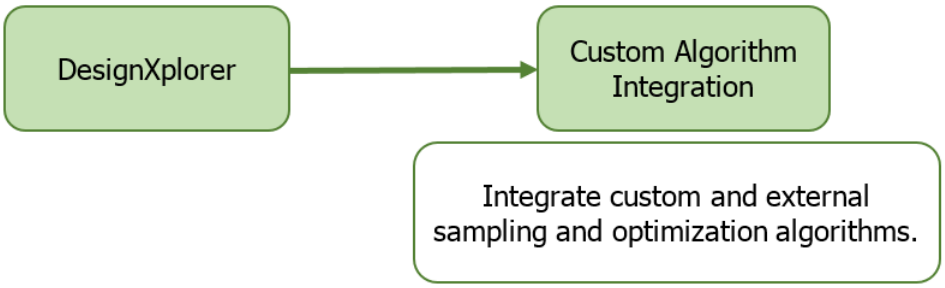
In the XML information, see these key XML elements:

```
<extension>
    <simdata>
      <Geometry>
```

Feature Creation in DesignXplorer

In DesignXplorer, you can integrate custom and external algorithms.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



In addition to the common capabilities listed in Feature Creation Roadmap (p. 25), DesignXplorer supports the feature creation capabilities that follow. References in this table take you to topics in the *ACT Customization Guide for DesignXplorer*.

Topic	Reference
Capabilities specific to DesignXplorer	DesignXplorer Feature Creation > • ACT-Based Properties for External Methods • DesignXplorer Extension Implementation • DesignXplorer Extension Capabilities
DesignXplorer – API usage examples	DesignExplorer APIs > • DesignExplorer APIs • Optimization APIs

The package **ACT Templates for DesignXplorer** provides five templates for integrating sampling and optimization methods. For more information, see ACT Templates for DesignXplorer (p. 87) in the *Ansys ACT Developer's Guide*.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

In the API information,see these DesignXplorer namespaces:

- **`Ansys.DesignXplorer.API`**
- **`Ansys.DesignXplorer.API.Common`**
- **`Ansys.DesignXplorer.API.Optimization`**
- **`Ansys.DesignXplorer.API.Sampling`**
- **`Ansys.DesignXplorer.API.Automation`**

In the XML information, see these key XML elements:

```
<extension>

    <simdata>
```

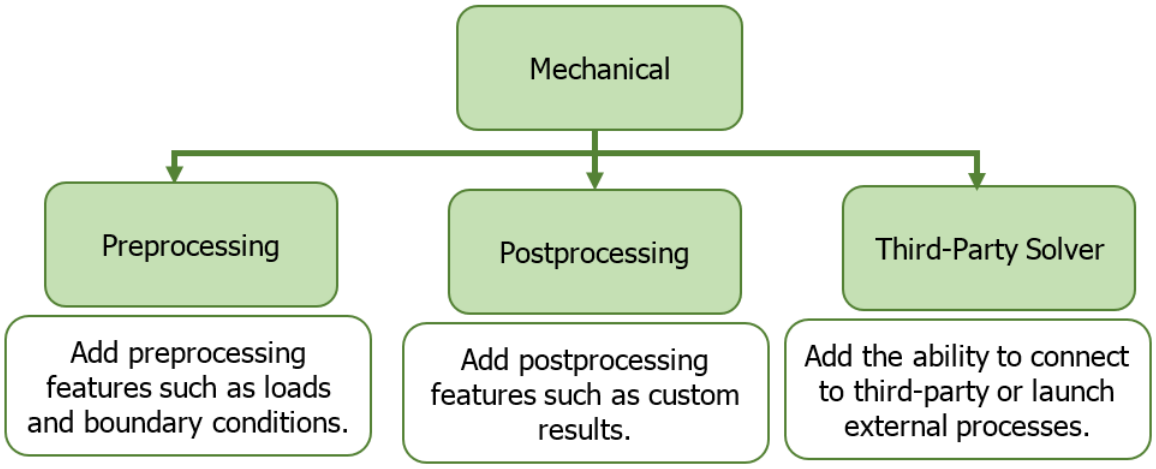

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

<extensionobject>

<optimizer>

Feature Creation in Mechanical

In Mechanical, you can create custom preprocessing, postprocessing, and third-party solver features.



In addition to the common capabilities listed in Feature Creation Roadmap (p. 25), Mechanical supports the feature creation capabilities that follow. References in this table take you to topics in the *ACT Customization Guide for Mechanical*. For information on Mechanical scripting and API usage, see *Scripting in Mechanical Guide*.

Topic	Reference
Capabilities specific to Mechanical	Mechanical Feature Creation > • UI Customization in Mechanical in the *ACT Customization Guide for Mechanical* • ACT-Based Property Parametrization in Mechanical • Preprocessing Capabilities in Mechanical • Postprocessing Capabilities in Mechanical • Third-Party Solver Connections in Mechanical • Additional Methods and Callbacks

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

In the API information, see these Mechanical namespaces:

- **Ansys.ACT.Automation.Mechanical**
- **Ansys.ACT.Automation.Mechanical.AnalysisSettings**

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

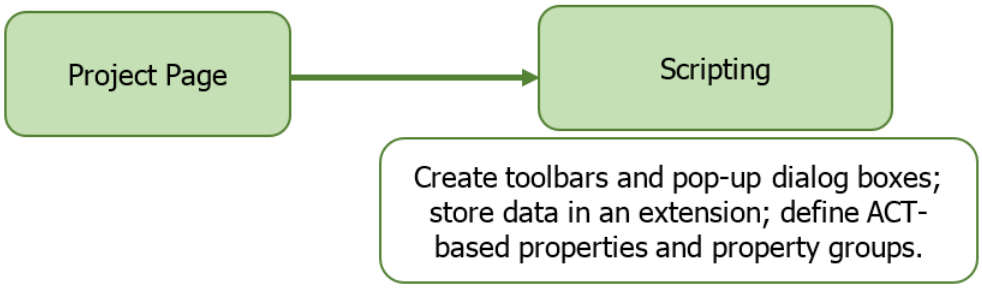
- **Ansys.ACT.Automation.Mechanical.BoundaryConditions**
- **Ansys.ACT.Automation.Mechanical.Connections**
- **Ansys.ACT.Automation.Mechanical.Enums**
- **Ansys.ACT.Automation.Mechanical.ImportedLoads**
- **Ansys.ACT.Automation.Mechanical.MeshControls**
- **Ansys.ACT.Automation.Mechanical.Results**
- **Ansys.ACT.Interfaces.Mechanical**
- **Ansys.ACT.Mechanical.Fields**

In the XML information, see these key XML elements:

```
<extension>
   <simdata>
      <load>
      <object>
      <result>
      <solver>
```

Feature Creation in the Workbench Project Page

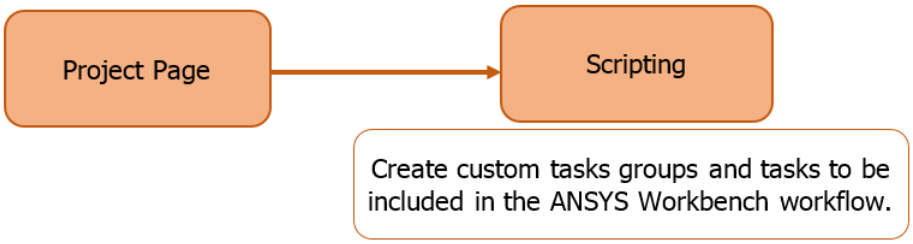
In the Workbench **Project** page, you can perform the customization activities common to products supporting feature creation. For more information, see the table in Feature Creation Roadmap (p. 25).



Simulation Workflow Integration Roadmap

Simulation workflow integration is the customization of the workflow in the Workbench **Project** page. You can create custom task groups (systems) and tasks (components) to facilitate interaction with the Workbench **Project Schematic**. References in this table take you to topics in the *ACT Customization Guide for Workbench*.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Topic	Reference
Capabilities specific to the Workbench **Project** page	Simulation Workflow Integration > • Custom Task Group and Task Exposure in the Project Schematic • Global Callbacks • Progress Monitoring • Process Utilities • Manual Creation of a Custom Workflow • XML Extension Definition for a Custom Workflow • IronPython Script for a Custom Workflow
Workbench **Project** Page – extension examples	Simulation Workflow Integration Examples > • Global Workflow Callbacks • Custom User-Specified Interface Operation • External Application Integration with Parameter Definition • External Application Integration with Custom Data and Remote Job Execution • Generic Material Transfer • Generic Mesh Transfer • Custom Transfer • Parametric Task Group
Appendices containing Workbench custom workflow information	• Appendix A: Component Input and Output Tables • Appendix B: Ansys-Installed System Component Template and Display Names • Appendix C: Data Transfer Types • Appendix D: Addin Data Types and Data Transfer Formats

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Topic	Reference
	• Appendix E: Ansys-Installed Custom Workflow Support

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide*.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

In the API information, see these simulation workflow namespaces:

- **`Ansys.ACT.Interfaces.Common`**
- **`Ansys.ACT.Interfaces.DataModel`**
- **`Ansys.ACT.Interfaces.UserInterface`**
- **`Ansys.ACT.Interfaces.UserInterface.Components`**
- **`Ansys.ACT.Interfaces.UserObject`**

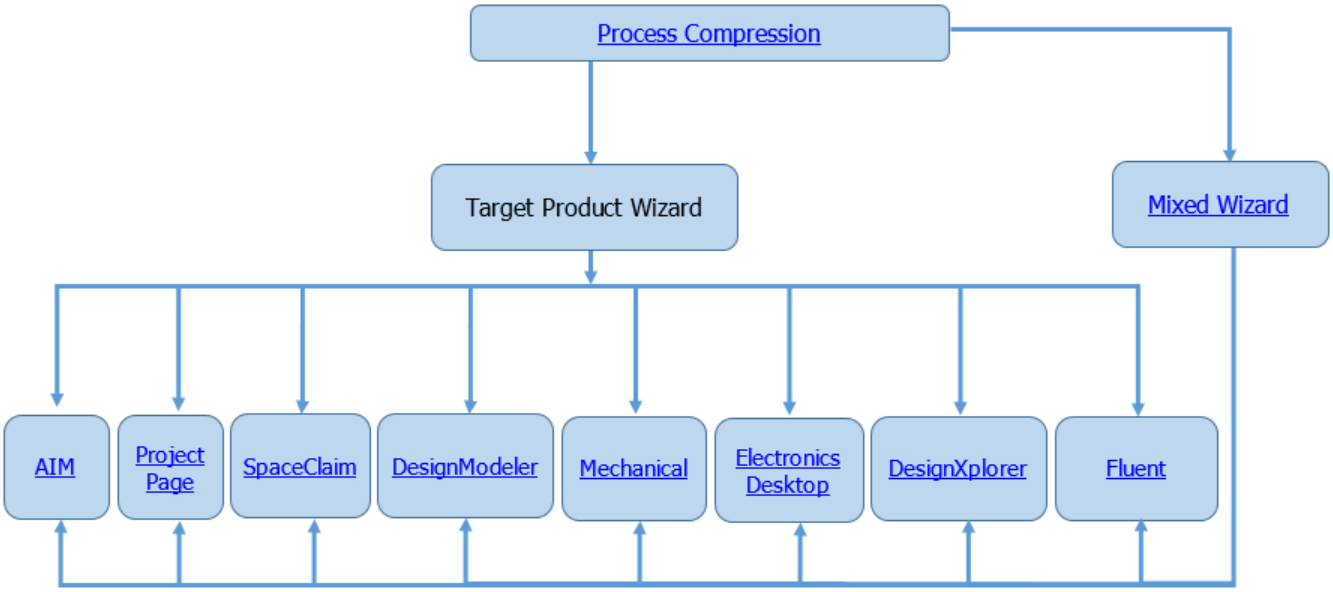
In the XML information, see these key XML elements:

```
<extension>
    <workflow>
        <callbacks>
        <property>
        <propertygroup>
        <propertytable>
        <taskgroups>
        <tasks>
```

Process Compression Roadmap

Process compression customization is the encapsulation and automation of existing processes available in an Ansys product into a simulation wizard (p. 141). The following roadmap shows the products for which ACT process compression capabilities are available.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Topics common to wizard creation follow. References in this table take you to topics in this guide.

Topic	Reference
General wizard information	Wizard Interface and Usage (p. 141)
Types of wizard (project, target product, and mixed)	Wizard Types (p. 143)
Creating wizards	Wizard Creation (p. 144)
Creating a mixed wizard	Mixed Wizard Example (p. 155)
Custom help files for wizards	Custom Wizard Help Files (p. 160)
Custom interfaces for Workbench-based wizards	Custom Wizard Interfaces (p. 162)
Custom interface example	Custom Wizard Interface Example (p. 164)

Note:

You use the Extension Manager (p. 41) to install and load extensions. You then use the Wizards launcher (p. 47) to start a wizard.

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide* and refer to the sections for the specific Ansys products being customized.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

In the API information, see this wizard namespace: **`Ansys.ACT.Interfaces.Wizard`**

In the XML information, see these key XML elements:

`<extension>`

`<application>`

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

<appstoreid>

<author>

<description>

<propertytable>

<uidefinition>

<wizard>

The following topics provide product-specific process compression information:

Process Compression in DesignModeler

Process Compression in DesignXplorer

Process Compression in Electronics Desktop

Process Compression in Fluent

Process Compression in Mechanical

Process Compression in SpaceClaim

Process Compression in the Workbench Project Page

Process Compression for Multiple Ansys Products (Mixed Wizards)

Process Compression in DesignModeler

You can create target product wizards to compress and automate processes in DesignModeler. The reference in the following table takes you to a topic in the *ACT Customization Guide for DesignModeler*. This wizard is part of the mixed wizard example (p. 155) described in this guide.

Topic	Reference
DesignModeler wizard example	DesignModeler Wizards

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide*.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Process Compression in DesignXplorer

Because DesignXplorer is part of Workbench, you can create project wizards and mixed wizards that can engage DesignXplorer. For the wizard step, you simply need to set the attribute **<context>** to **Project**. Because DesignXplorer does not have a stand-alone application interface, you cannot create target product wizards for DesignXplorer.

For more information about project wizards and mixed wizards for Workbench, see:

- Process Compression in the Workbench Project Page (p. 36)
- Process Compression for Multiple Ansys Products (Mixed Wizards) (p. 36)

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Process Compression in Electronics Desktop

You can create target product wizards to compress and automate processes in Electronics Desktop. Wizards can be run in Electronics Desktop when it is opened from Workbench or when it is opened as a standalone instance.

The reference in the following table takes you to a topic in the *ACT Customization Guide for SpaceClaim*.

Topic	Reference
Electronics Desktop wizard example	Electronics Desktop Wizards

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide*.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Process Compression in Fluent

You can create target product wizards to compress and automate processes in Fluent. Wizards can be run in Fluent when it is opened from Workbench or when it is opened as a standalone instance.

References in the following table take you to topics in the *ACT Customization Guide for Fluent*.

Topic	Reference
Fluent wizard examples	Fluent Wizards > • Fluent Wizard (MSH Input File) • Fluent Wizard (CAS Input File)

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide*.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Process Compression in Mechanical

You can create target product wizards to compress and automate processes in Mechanical. The reference in the following table takes you to a topic in the *ACT Customization Guide for Mechanical*. This wizard is part of the mixed wizard example (p. 155) described in this guide.

Topic	Reference
Mechanical wizard example	Mechanical Wizards

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide* and refer to the "Mechanical" section.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Process Compression in SpaceClaim

You can create target product wizards to compress and automate processes in SpaceClaim. Wizards can be run in SpaceClaim when it is opened from Workbench or when it is opened as a standalone instance.

References in the following table take you to topics in the *ACT Customization Guide for SpaceClaim*. The first reference takes you to a topic describing the SpaceClaim wizard that is part of the mixed wizard example (p. 155) described in this guide.

Topic	Reference
SpaceClaim wizard examples	SpaceClaim Wizards > • SpaceClaim Wizard for Building a Bridge • Space Claim Wizard for Generating a Ball Grid Assembly (BGA)

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide*.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Process Compression in the Workbench Project Page

You can create project wizards to orchestrate workflows in the Workbench **Project** page. The reference in the following table takes you to a topic in this guide.

Topic	Reference
Workbench project wizard example	Wizard Creation (p. 144)

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide* and refer to the "Workbench" section.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Process Compression for Multiple Ansys Products (Mixed Wizards)

You can create a mixed wizard, which executes across multiple Ansys products. A mixed wizard begins in the Workbench **Project** page and engages any data-integrated product with Workbench journaling and scripting capabilities, such as DesignModeler or Mechanical.

The reference in the following table takes you to a topic in this guide.

Topic	Reference
Workbench mixed wizard example	Mixed Wizard Example (p. 155)

For journaling and scripting information, including product-specific data container details, see the *Workbench Scripting Guide* and refer to the sections for the products in which wizards are executed.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

For comprehensive API and XML information, see the *Ansys ACT API and XML Online Reference Guide*.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.

ACT Tools

ACT provides many tools for developing, debugging, and executing extensions. To use these tools effectively, you must first understand how extensions are created (p. 71) and debugged (p. 169).

Tool	Description
ACT Start Page (p. 39)	The **ACT Start Page** is a single page that provides you with convenient access to ACT functionality. From this page, you can access tools for developing, debugging and executing extensions.
Extension Manager (p. 41)	The **Extension Manager** is for installing and loading extensions. Extensions cannot be executed until they are installed and loaded.
Wizards Launcher (p. 47)	The **Wizards** launcher is for starting a simulation wizard (p. 141). After an extension with a wizard is installed and loaded, you use the **Wizards** launcher to start the wizard.
Extensions Log File (p. 47)	The **Extensions Log File** displays messages generated by extensions. If warnings are present, they display in orange. If errors are present, they display in red.
Binary Extension Builder (p. 49)	The binary extension builder creates a compiled WBEX file from a scripted extension that can be shared with others. Once you finish developing and debugging a scripted extension, you can compile it into a binary extension. The contents of a binary extension cannot be viewed or edited.
ACT Console (p. 53)	The **ACT Console** exposes the ACT API so that you can interactively test commands as you develop and debug scripts.
ACT App Builder (p. 94)	The **ACT App Builder** is for a creating ACT extensions in a visual environment, saving you from having to create or modify XML files and IronPython scripts directly.
ACT Debugger (p. 170)	The **ACT Debugger** extends the **ACT Console** to provide state-of-the-art debugging capabilities. You can use it or Microsoft® Visual Studio (p. 181) for debugging.
ACT Workflow Designer	The **ACT Workflow Designer** is for automating the creation of custom simulation workflows on the Workbench **Project** page. For more information, see the ACT Customization Guide for Workbench.

ACT Start Page

The **ACT Start Page** is available in all Ansys products that support ACT extensions. See the introduction (p. 1) for a list of these products.

This guide assumes that you are accessing this page from Workbench. For access information to this page from stand-alone instances of Electronics Desktop, Fluent, and SpaceClaim, see the ACT customization guides for these products. The concluding section (p. 183) provides links to these guides.

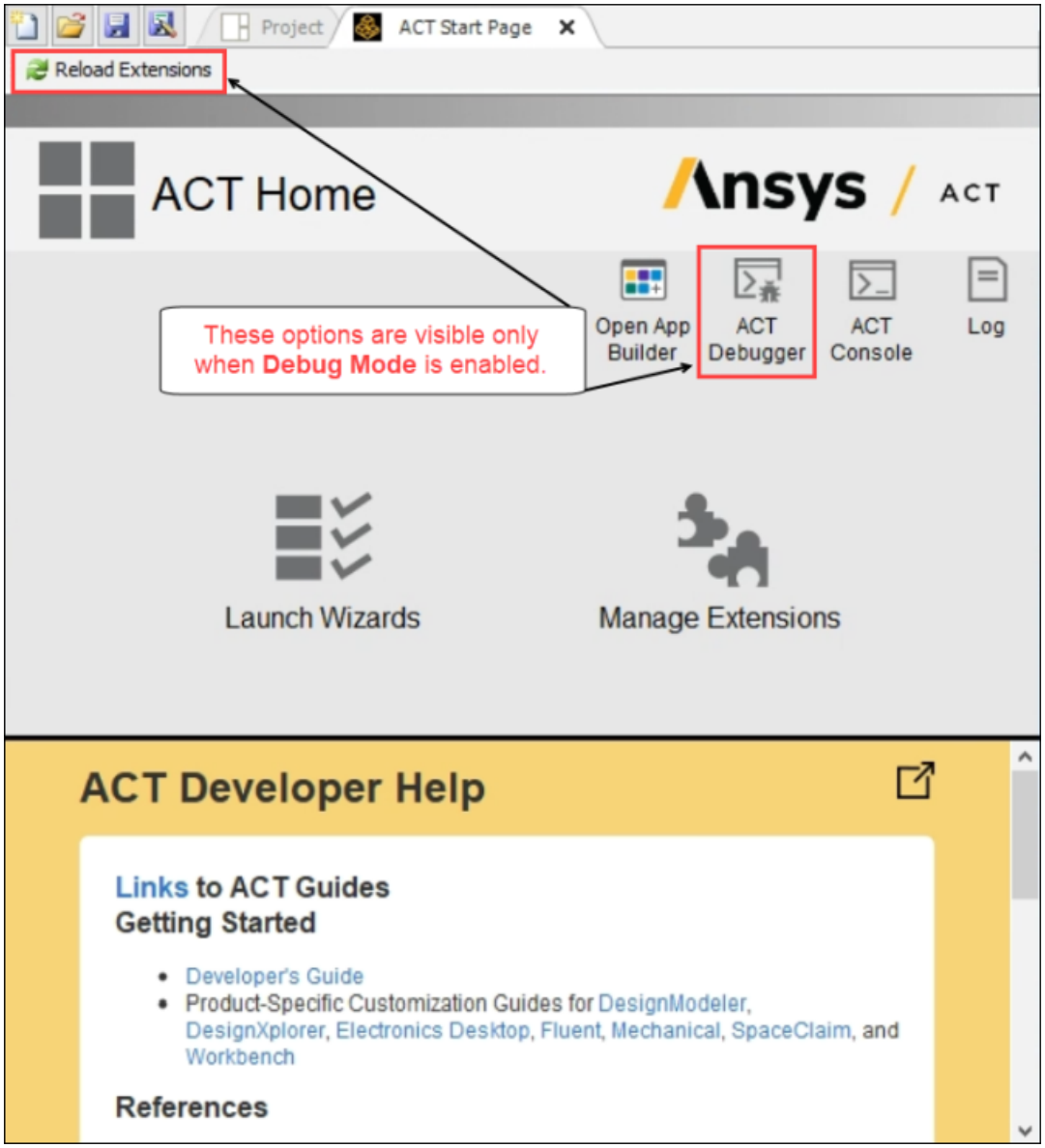
To access the **ACT Start Page**, do one of the following:

- Click **ACT Start Page** in the toolbar.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Select **Extensions** → **ACT Start Page**.

The **ACT Start Page** displays icons for accessing many ACT tools. The **Extensions** menu also provides options for accessing these tools.



The lower help panel provides links to ACT guides, supplied examples (p. 81), and release-specific documents. Clicking the resize button in the upper right corner opens this panel in a new window in your browser.

Note:

On the Ansys Store, the App Developer Resources page displays tabs with links to not only these ACT documents but also many more.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extension Manager

The **Extension Manager** is for installing and loading extensions. You cannot execute an extension until it is installed and loaded.

There are two versions of the **Extension Manager**, both of which offer similar capabilities:

- To access the graphic-based version, click **Manage Extensions** on the **ACT Start Page**.
- To access the table-based version, select **Extensions** → **Manage Extensions**.

The **Extension Manager** displays all extensions that are installed and available for loading. To define additional folders in which ACT is to search for extensions, you use the Additional Extension Folders (p. 79) option. This option is available when you click the gear icon in the graphic-based **Extension Manager** and in **Tools** → **Options** → **Extensions**.

Note:

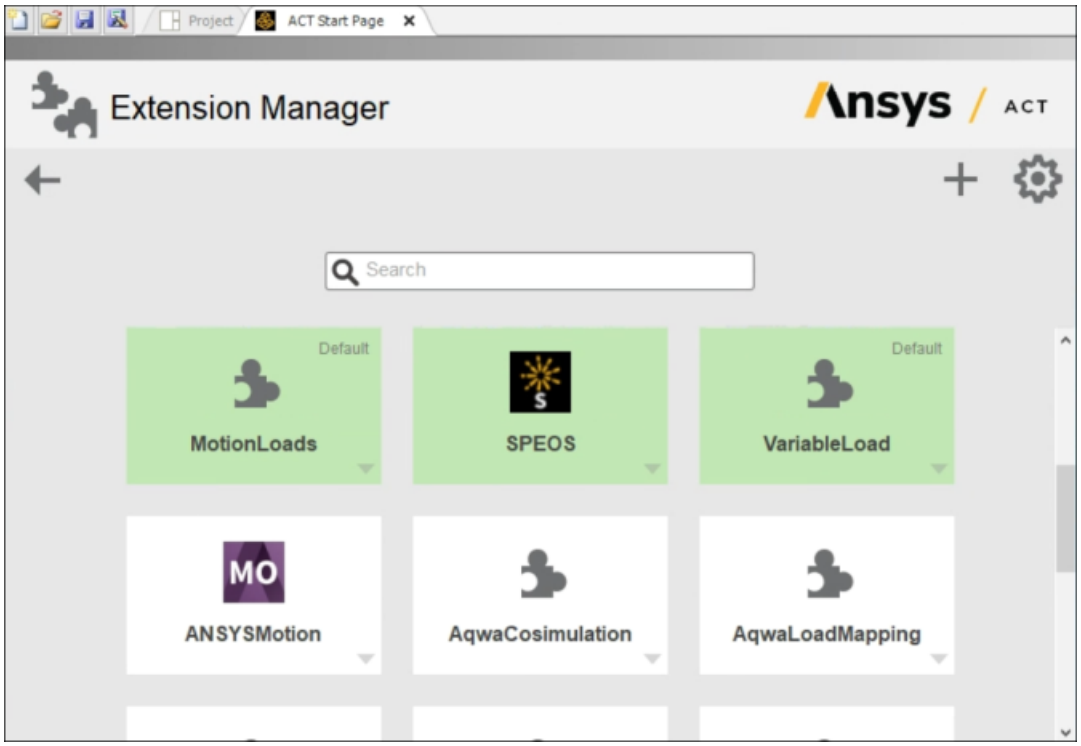
While the **Extension Manager** always lists installed binary extensions, it lists installed scripted extensions only if you have an ACT license.

The following topics describe how to use both versions of the **Extension Manager**:

Using the Graphic-Based Extension Manager

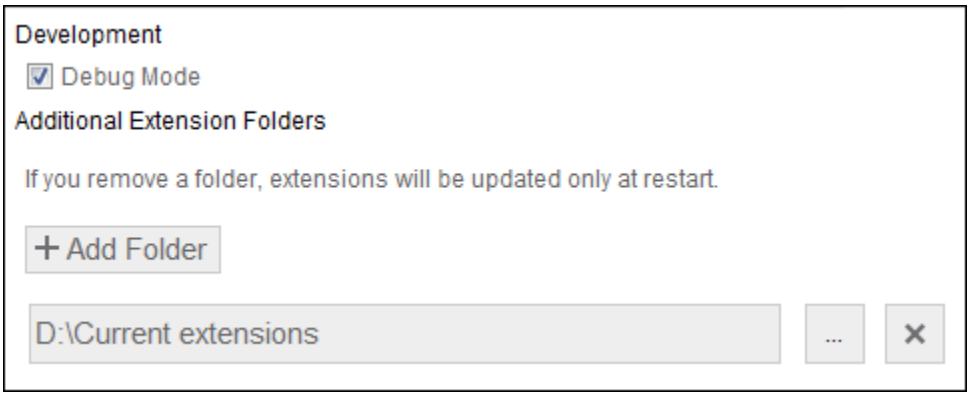
The following figure displays the graphic-based **Extension Manager** that is accessed by clicking **Manage Extensions** on the **ACT Start Page**. As noted earlier, installed scripted extensions are shown only if you have an ACT license. In this figure, many extensions are installed. When an extension is loaded, the block background is green rather than white.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



In the graphic-based **Extension Manager**, you can do the following:

- Click one of the plus (+) icons to install a new binary extension.
- Click the gear icon to access Debug Mode (p. 81) and Additional Extension Folders (p. 79) options. These options serve the same purpose as those in **Tools** → **Options** → **Extensions**. However, rather than having to manually define the folders that the **Extension Manager** searches for the extensions to display, here you click **Add Folder** and then browse to and select the folder.



- Click **Ansys Store** in the lower left corner to explore ACT apps available for download.
- Click the left-pointing arrow in the upper left corner to return to the **ACT Start Page**.

Right-clicking an extension displays a context menu with the options available for the extension. The options shown depend on the current state of the extension. Descriptions follow for all possible options:

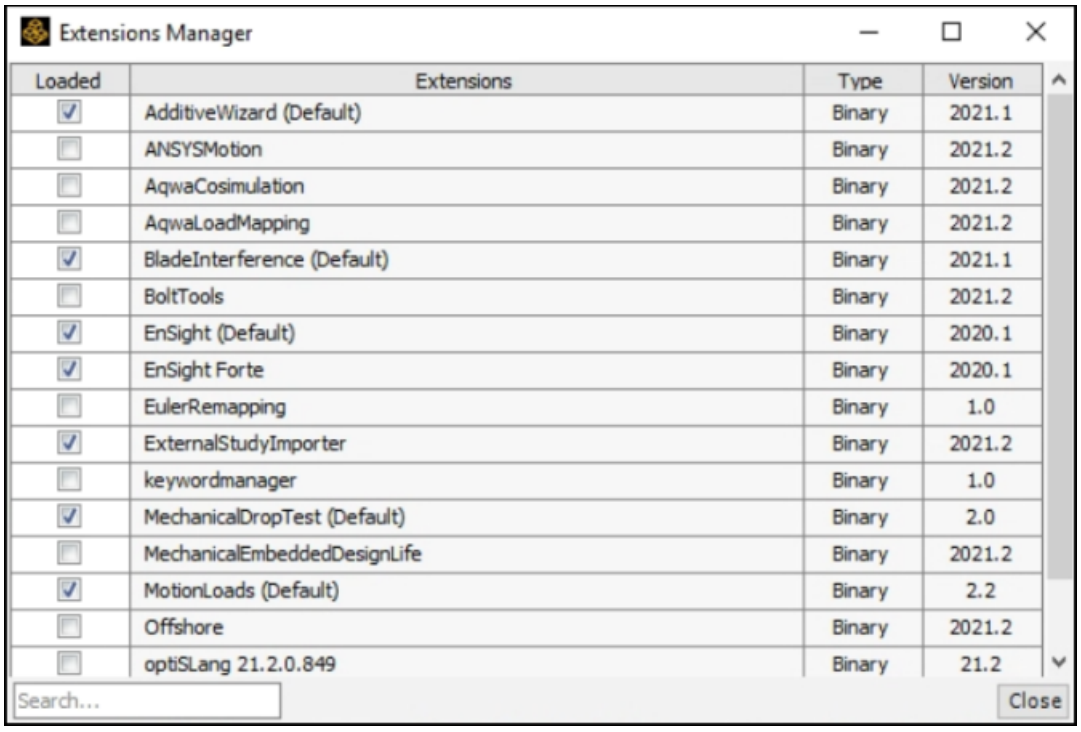
- **Load extension**: Available only for an unloaded extension, this option loads the extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- **Unload extension**: Available only for a loaded extension, this option unloads the extension
- **Load as default**: Available only for an extension that is not automatically loaded to a project by default, this option loads the extension to either an existing or new project when Workbench or the corresponding targeted product is started. For more information, see Configuring Extensions to Load by Default (p. 46).
- **Do not load as default**: Available only for an extension that is automatically loaded to a project by default, this option does not load the extension to either an existing or new project when Workbench or the corresponding targeted product is started.
- **Uninstall**: Available only for unloaded binary extensions, this option uninstalls the binary extension.
- **Build**: Available only for scripted extensions, this option opens the binary extension builder (p. 49).
- **About**: Always available, this option displays extension information such as the version, format (`xml` or `wbex`), folder, guid, context, and feature.

Using the Table-Based Extension Manager

The following figure displays the table-based **Extension Manager** that is accessed by selecting **Extensions** → **Manage Extensions**. As noted earlier, installed scripted extensions are shown only if you have an ACT license. In this figure, many extensions are installed. To load an extension, you select its check box in the **Loaded** column.



Right-clicking an extension displays a context menu with the options available for the extension. The options shown depend on the current state of the extension. They are comparable to the options described in the previous topic for the graphic-based **Extension Manager**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Searching for an Extension

The **Extension Manager** provides a **Search** option for finding a specific extension. The case-insensitive search option looks for all entered strings that are separated by a blank, which serves as the AND operation. The search option performs a combined search for strings separated by the OR operation. You can search by the following items:

Extension name, description, or author

Example 1: Type **`mydemowizards`** to return all extensions with this name.

Example 2: Type **`authorname`** to return all extensions with **`author`** set to **`authorname`**.

Context

Example: Type **`Mechanical`** to return all extensions with **`context`** set to **`Mechanical`**.

Extension object type

Example: Type an object type to return all extensions containing at least one object of this type. For instance, type **`wizard`**, **`workflow`**, or any object for the element **`simdata`**, such as **`load`**, **`result`**, **`solver`**, or **`geometry`**.

Extension object type and object name separated by a colon

Example: Type **`load:my_load`** to return all extensions with a **`load`** named **`my_load`**.

Installing and Uninstalling Extensions

The processes differ for installing and uninstalling scripted extensions versus binary extensions:

Installing and Uninstalling Scripted Extensions

Installing and Uninstalling Binary Extensions

Note:

When Workbench is not installed, the installation location for an extension with a Fluent or SpaceClaim wizard differs. For more information about stand-alone instances, see the ACT customization guides for these two products. The concluding section (p. 183) provides links to these guides.

Installing and Uninstalling Scripted Extensions

The **Extension Manager** displays all installed scripted extensions when an ACT license is available.

To install a scripted extension, save the extension and associated files in one of the following locations:

- **`%ANSYSversion_DIR%\..\Addins\ACT\extensions`**
- **`%APPDATA%\Ansys\v222\ACT\extensions`**

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- Any of the additional directories specified for Additional Extension Folders (p. 79). This option is available when you click the gear icon in the graphic-based **Extension Manager** and in **Tools** → **Options** → **Extensions**.

To uninstall a scripted extension, remove the extension and its associated files and folders. If you uninstall a scripted extension while the **Extension Manager** is open, it continues to display this extension until you close and reopen the **Extension Manager**.

Note:

The **Uninstall** option is available on the right-click context menu only for a binary extension.

Installing and Uninstalling Binary Extensions

The **Extension Manager** displays all binary extensions (WBEX files) that are installed, regardless of whether an ACT license is available.

To install a binary extension:

1. Access the install functionality by doing one of the following:
   - Select **Extensions** → **Install Extension**.
   - In the graphic-based **Extension Manager** accessed from the **ACT Start Page**, click one of the plus (+) icons.
2. In the **Open** dialog box that appears, navigate to and select the WBEX file that you want to install and click **Open**.

Installed binary extensions are installed to your application data folder (**`%APPDATA%\Ansys\v222\ACT\extensions`**) and are available for loading in the **Extension Manager**.

Note:

When you install a binary extension, a new folder and WBEX file are created because both are necessary for compatibility with ACT. If you need to move the extension to a different folder, make sure that both the folder and WBEX file are copied to the same folder at the same time.

To uninstall a binary extension, simply right-click it and select **Uninstall**, which is available only for binary extensions.

Caution:

When uninstalling a binary extension, ACT permanently deletes both the definition file and folder. You will lose all changes that you have made to the extension's folder contents. Best practice is to use your application data folder

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

%APPDATA%\Ansys\v222\ACT\extensions only for installing WBEX files and another folder for developing your source code.

Loading and Unloading Extensions

In the **Extension Manager**, you load and unload an installed extension by right-clicking the extension and selecting the appropriate option. When an extension is marked as loaded, it is loaded with the corresponding targeted product specified by the attribute **context** for the extension.

- In the graphic-based **Extension Manager** accessed from the **ACT Start Page**, you can also load or unload an extension by simply clicking the block for the extension.
- In the table-based **Extension Manager** accessed from the **Extensions** menu, you can also load or unload an extension by selecting or clearing the check box to the left of the extension name.

Note:

- Loading is automatic for extensions that have already been loaded and saved to the project. Any extension to be automatically loaded must be available to the **Extension Manager**. If the **Extension Manager** cannot load a required extension when a project is opened, a warning message appears, indicating that you might encounter limited behavior if you proceed. Clicking **Show Details** in this message lists the extensions that are missing from the project. Clicking **OK** proceeds with opening the project.
- If you use either version of the **Extension Manager** in Workbench to load an extension with the attribute **context** set to **Mechanical**, then Mechanical knows that this extension is loaded. However, if you unload the extension using either version of the **Extension Manager** in Workbench, then Mechanical does not know that the extension has been unloaded.

Configuring Extensions to Load by Default

In the **Extension Manager**, you configure an extension to load to the project by default by right-clicking it and selecting **Load as Default**. An extension loaded by default is automatically loaded when the corresponding targeted product is started. No limit exists on the number of extensions that can be loaded by default.

An extension that is configured to load by default displays the word **Default**.

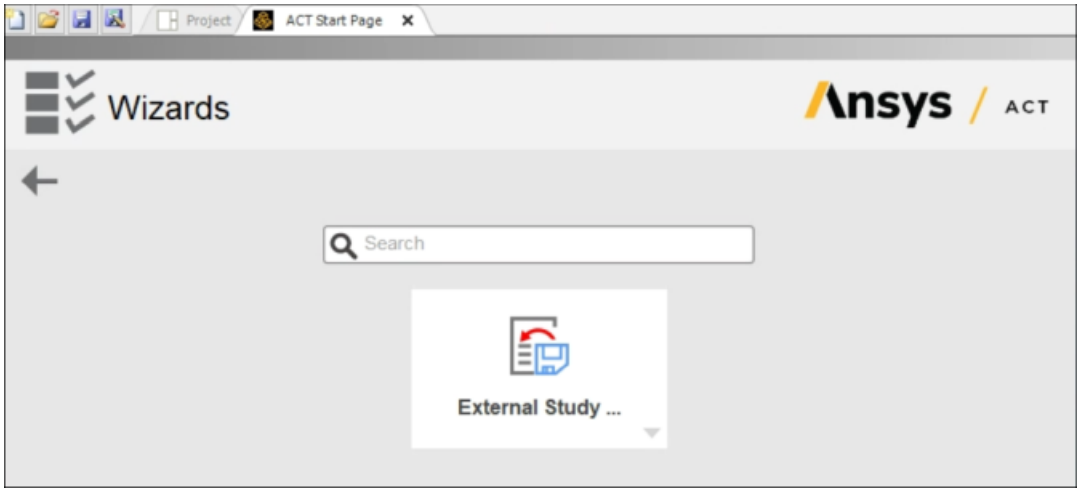
- In the graphic-based **Extension Manager** accessed from the **ACT Start Page**, the word **Default** appears in the upper right corner of the block for the extension.
- In the table-based **Extension Manager** accessed from the **Extensions** menu, the word **Default** appears in parentheses after the extension name.

If you no longer want an extension to load by default, right-click it and select **Do Not Load as Default**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Wizards Launcher

The **Wizards** launcher is for starting a simulation wizard (p. 141). When you click **Launch Wizards** on the ACT Start Page (p. 39), the **Wizards** launcher displays all loaded extensions that have wizards for the current context.



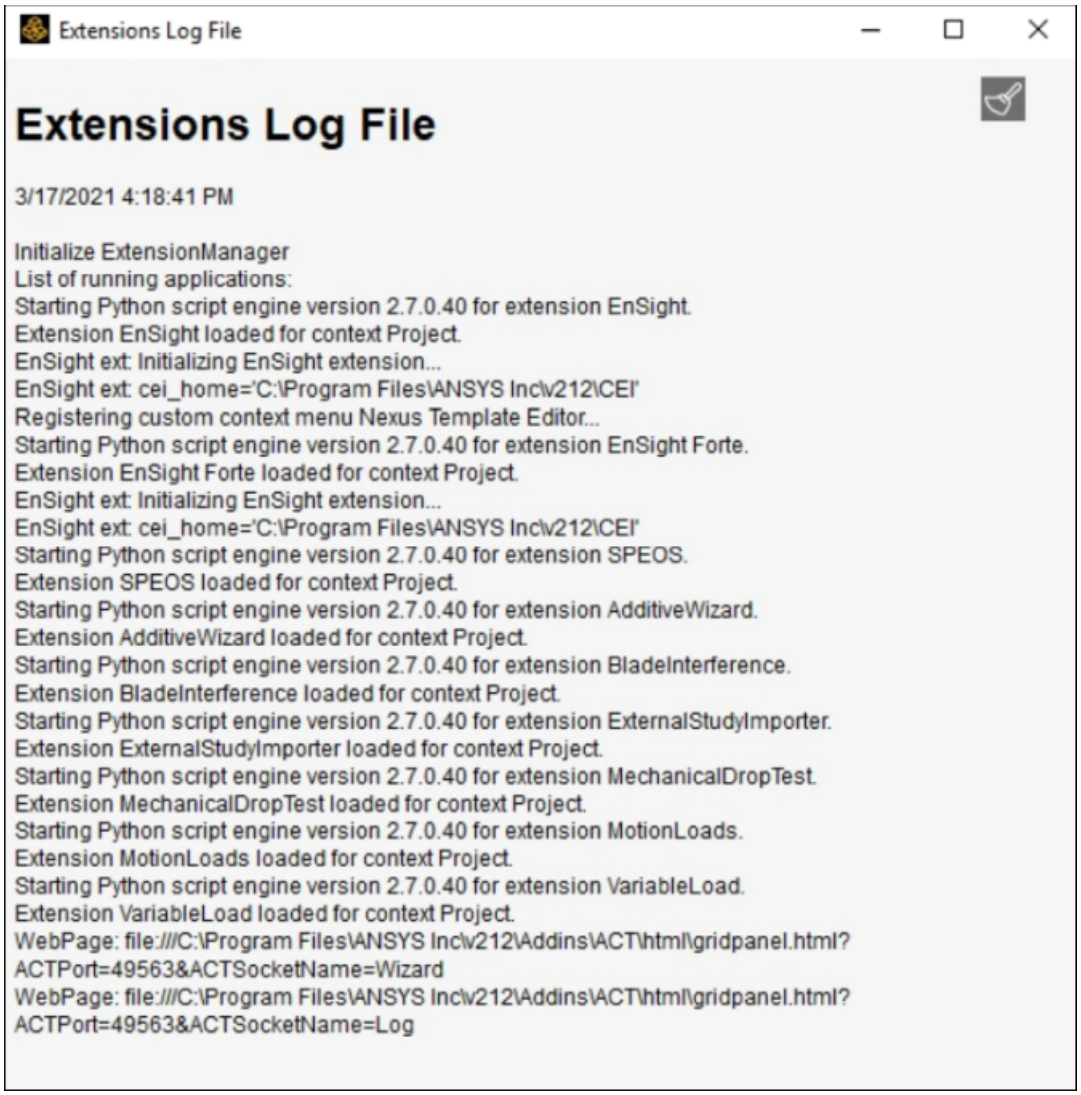
The **Search** option here is the same as the one in the **Extension Manager**. For more information, see Searching for an Extension (p. 44).

To start a wizard, click the block for the extension. Optionally, right-click it and select **Execute Wizard**. The wizard starts, showing the first step.

Extensions Log File

The **Extensions Log File** displays messages generated by extensions. If warnings are present, they display in orange. If errors are present, they display in red.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



You can open the **Extensions Log File** from many different places:

In Workbench:

You can open the **Extensions Log File** in one of two ways:

- On the **ACT Start Page**, click the **Log** button in the toolbar.

- Select **Extensions** → **View Log File**.

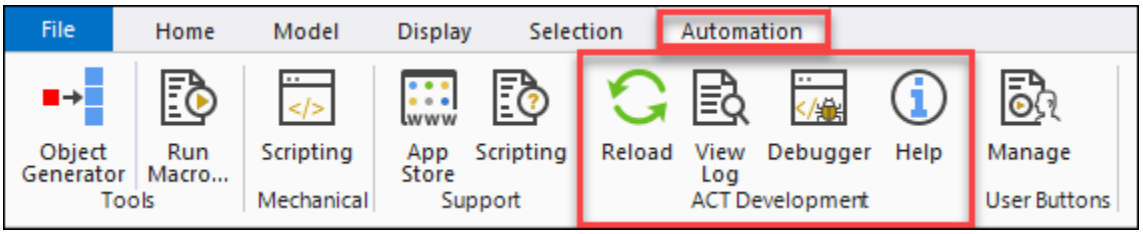
In DesignModeler:

From the **ACT Development** toolbar, click the button for switching between opening and closing the **Extensions Log File**. This toolbar is available only when the Debug Mode (p. 81) check box is selected in **Tools** → **Options** → **Extensions**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

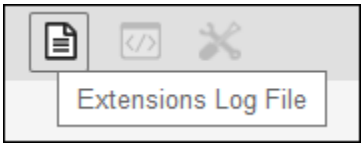
In Mechanical:

In the **ACT Development** group on the ribbon's **Automation** tab, click **View Log**. This group is available only when the Debug Mode (p. 81) check box is selected in **Tools** → **Options** → **Extensions**.



In the App Builder:

In the toolbar for the ACT App Builder (p. 94), click the button for the **Extensions Log File**.



You can resize and drag the window displaying the **Extensions Log File**. You can also keep this window open as you switch between products. To clear the **Extensions Log File**, you click the brush icon in the window's upper right corner.

Tip:

To open the **Extensions Log File** in your web browser, you can copy all of the text that follows **WebPage:** and paste it into your browser's address bar. For example, given the log file shown in this topic, you'd copy and paste the following text:

```
file:///C:\Program Files\ANSYS Inc\v212\Addins\ACT\html\gridpanel.html?ACTPort=49563&ACTSocketName=Log
```

Binary Extension Builder

The binary extension builder creates a compiled WBEX file from a scripted ACT extension that can be shared with others. A separate compiler is not necessary for building a binary extension. ACT provides its own process for encapsulating all folders and files necessary for the extension into the WBEX file, which cannot be viewed or edited.

The appearance of the binary extension builder depends on how it was accessed. The following topics provide access and usage information:

Accessing the Binary Extension Builder

Building a Binary Extension

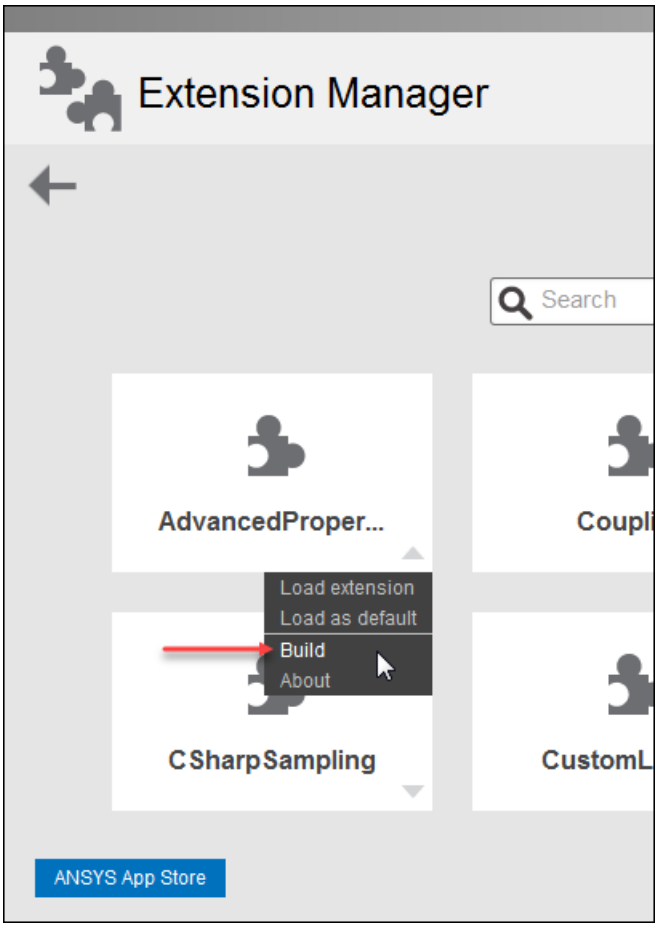
Compiling Multiple IronPython Scripts

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

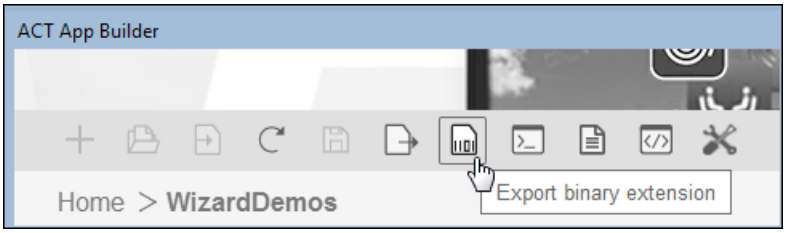
Accessing the Binary Extension Builder

You can access the binary extension builder using any of the following methods:

- Select **Extensions** → **Build Binary Extension**.
- In the graphic-based **Extension Manager** accessed form the **ACT Stat Page**, right-click the extension and select **Build**.



- In the ACT App Builder (p. 94), when an app builder project is open, click the toolbar button for exporting a binary extension.



Building a Binary Extension

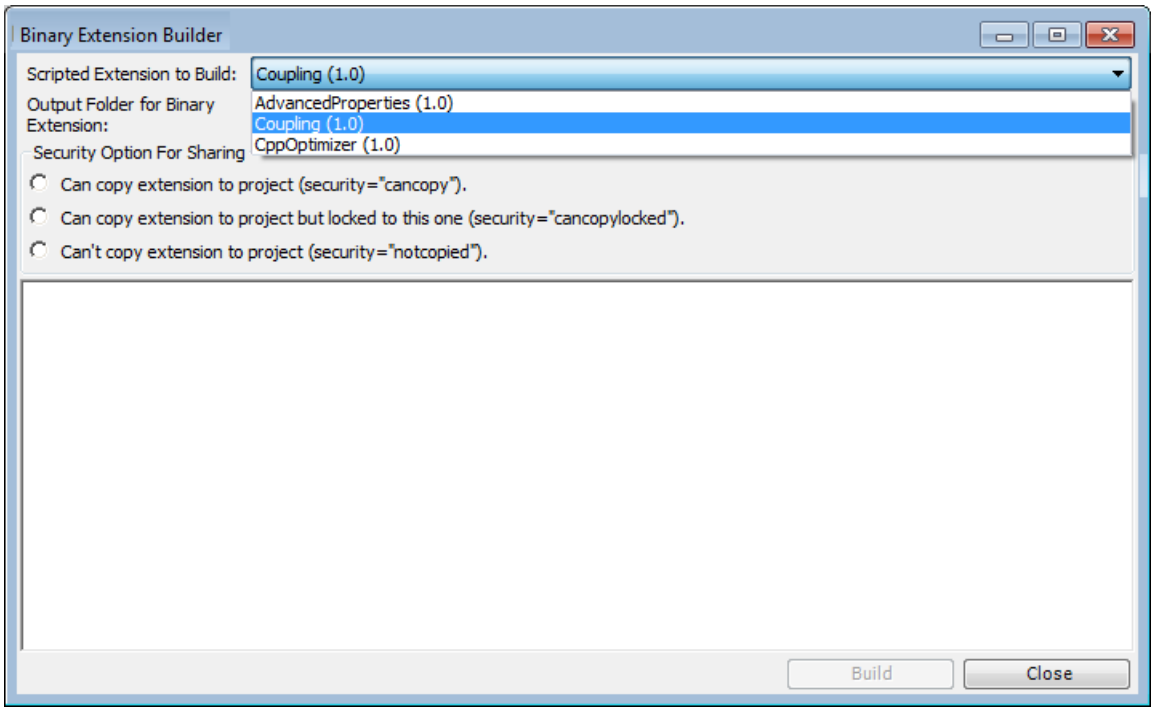
To build a binary extension from a scripted extension, do the following:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

1. If the binary extension builder was accessed from the **Extensions** menu, for **Scripted Extension to Build**, select the scripted extension to build as a WBEX file.

   Choices include all scripted extensions in directories specified for Additional Extension Folders (p. 79). This option is available when you click the gear icon in the graphic-based **Extension Manager** and in **Tools** → **Options** → **Extensions**.

   Both the extension name and version are shown to avoid confusion in case multiple versions of an extension are defined.



   If the binary extension builder is accessed using any other method, **Scripted Extension to Build** is not shown because the scripted extension is already selected.

2. For **Output Folder for Binary Extension**, identify the folder in which to output the WBEX file.

3. For **Security Option For Sharing**, select a security level.

   Your selection specifies whether the extension can be saved within an Workbench project and, when that project is shared, whether the extension can be loaded with the shared project. The security level allows the developer of the extension to control how the extension can be shared and used with various projects. Choices are:

   - **Can copy extension to project**: Each time a user asks to save the extension with a project, the extension itself is copied into the project and consequently is available each time the project is opened. The extension can also be used with other projects.
   - **Can copy extension to project but locked to this one**: The extension can be saved within a project, as with the previous option, but the use of the extension is limited to the current project. If a user opens a new project, the extension is not available.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- **Can't copy extension to project**: The extension is not saved with the project and is not shared with other users of the project.

Note:

- The extension can be sent separate from a project. The process for saving extensions within a project is described in Configuring Extensions to Load by Default (p. 46).
- Security options only apply when users attempt to save the extension with a project. Otherwise, they are not applicable.

Once all options are specified, the **Build** button is enabled.

4. Click **Build**.

While the WBEX file is being built, the bottom part of the window displays build information.

Compiling Multiple IronPython Scripts

The following excerpt shows how the XML file for a scripted extension can reference multiple IronPython scripts:

```
<extension version="2" minorversion="1" name="WizardDemos">
 <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
 <author>Ansys, Inc.</author>
 <description>Simple extension to test wizards in different contexts.</description>

 <script src="main.py" />
 <script src="ds.py" />
 <script src="dm.py" />
 <script src="sc.py" />
```

The element **`<script>`** appears four times. All four of these scripts are placed in the extension folder. When ACT processes multiple scripts, it loads them into a single scope, as if the contents of all scripts are contained in a single flat file. This works for a scripted extension that uses **`import`** statements because all scripts reside in the extension folder.

Before using the binary extension builder to compile a scripted extension into a WBEX file, you can specify which scripts to compile by setting the attribute **`compiled`** to **`true`** for the element **`<script>`**:

```
 <script src="main.py" compiled="true" />
 <script src="ds.py" compiled="true" />
 <script src="dm.py" compiled="true" />
 <script src="sc.py" compiled="true" />
```

When multiple scripts are marked as **`compiled`**, the binary extension essentially pushes all of the content in these scripts into the binary buffer stream. When you install and load the WBEX file, ACT reads each script from the binary buffer and loads the script content into a single scope.

Consequently, the result for the binary version is the same as the scripted version. All contents in the multiple scripts are loaded into a single scope without any module designations, just as if you had originally combined the different scripts into one large, single script.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Unlike a scripted extension, the installed binary extension can no longer import another script as a module because the scripts no longer reside in an extension folder. Because methods and classes are invoked as if all scripts are in one large, single script, you must remove import statements and module prefixes before building a binary extension. By flattening the scripts in this way, both the scripted and binary versions of the extension run successfully.

This flattening does make it difficult to test the scripted extension before building the binary version because you cannot truly test the implementation until after you create and install the WBEX file. However, you can compile your IronPython modules into DLLs, remove the corresponding elements for the scripts from the XML file, and import them as required in your main script.

Note:

You cannot currently flag files to have the binary extension builder skip them. During the building of a WBEX file, any messages that you see about files being skipped are the result of ACT no longer needing the plain-text scripts because the elements for the scripts are marked as **`compiled="true"`**.

ACT Console

The **ACT Console** exposes the ACT API so that you can interactively test commands as you develop and debug scripts. Currently, the ACT API provides coverage for the following Ansys products:

- Workbench
- DesignModeler
- Mechanical

The following topics describe the ACT API, the console, and how to use it:

Understanding the ACT API

ACT supports customization of Ansys products with a set of interfaces. The high-level member interfaces of the API expose the properties and methods that allow access to the underlying product data.

The extensions that are loaded by the Extension Manager (p. 41) are each configured and initialized to support Python scripting. For each extension, the global variable **`ExtAPI`** gives access to the property **`DataModel`**. This property returns the object **`IDataModel`**, which has properties that

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

return all high-level interfaces available for customizations. The complete list of member interfaces is given in the *Ansys ACT API Reference Guide*.

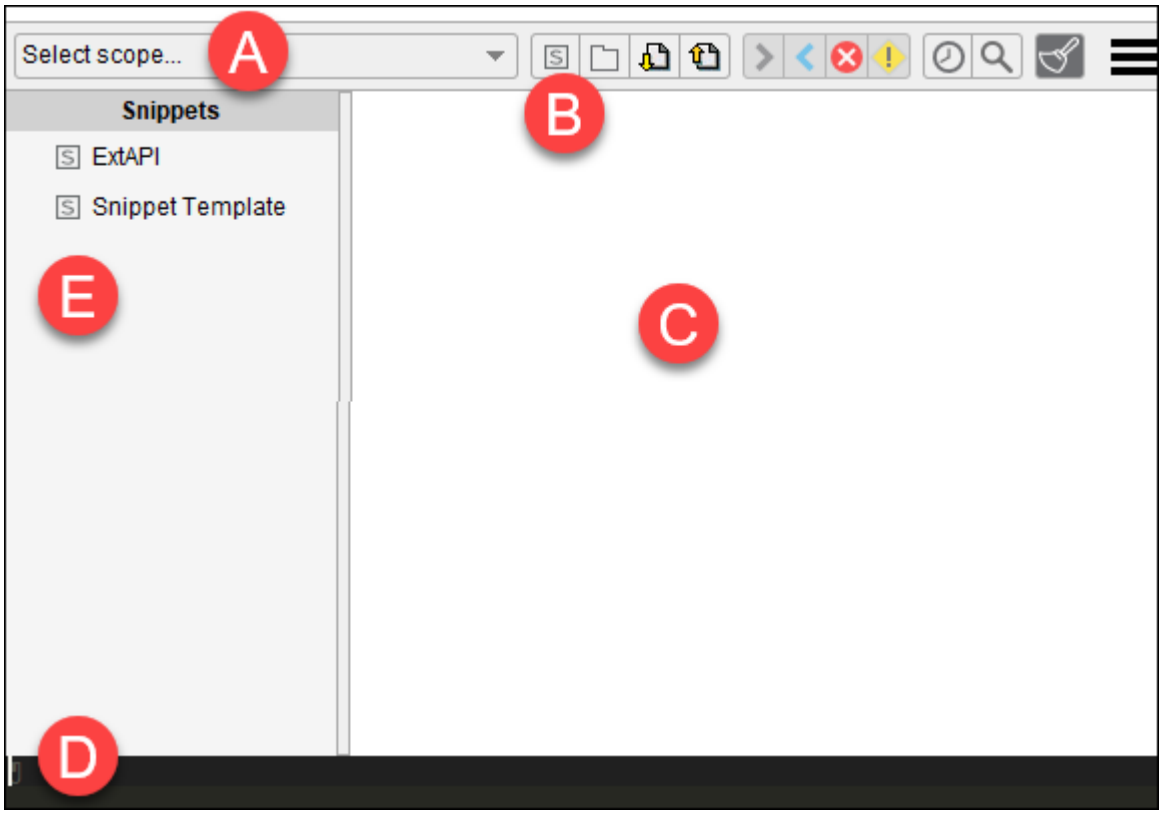
Each Ansys product can leverage its own scripting language and define unique object API. ACT minimizes inconsistency and confusion introduced by product-specific syntax by providing its own *Automation API*. Automation objects and methods wrap underlying product API by re-exposing user-facing features in a clear and consistent syntax. Instead of piecemeal script statements for cross-product customization, you can program with one ACT language.

You typically engage automation-level objects when issuing commands inside the console or invoking **`ExtAPI.DataModel`** members. You can confirm automation API usage by executing a variable name in the console. If the returned string starts with **`Ansys.ACT.*.Automation.`**, you are working with the automation API. ACT substitutes the appropriate product name in place of the asterisk (such as **`Ansys.ACT.WorkBench.Automation.`**).

Conversely, you typically encounter **`SimEntity`**-level objects as arguments to extension callbacks.

Understanding Console Components

The following of the **ACT Console** indicates console components. You can resize and drag the console window. You can also keep it open as you switch between products.



Callout	Name	Description
A	Scope	If you have ACT extensions loaded, you can select the one with which you want to interact. However, interacting with ACT extensions is not required. If no extension is selected, you interact with the console generally. For more information, see Selecting the Scope (p. 56).

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Callout	Name	Description
B	Toolbar	Buttons for managing snippets and the command history. For more information, see Snippet Toolbar Buttons (p. 64) and Snippet Toolbar Buttons (p. 64).
C	Command History	Shows previously entered commands and any information returned from each command. For more information, see Working with Command History Entries (p. 56).
D	Command Line	Begin entering commands to see autocomplete options in the tooltip. Use Ctrl + ↑ to access previous commands.For more information, see Entering Commands in the Command Line (p. 56).
E	Snippets	Stores commands or blocks of code that you use repeatedly. For more information, seeUsing Snippets (p. 62).

In Workbench:

You can open the console in one of two ways:

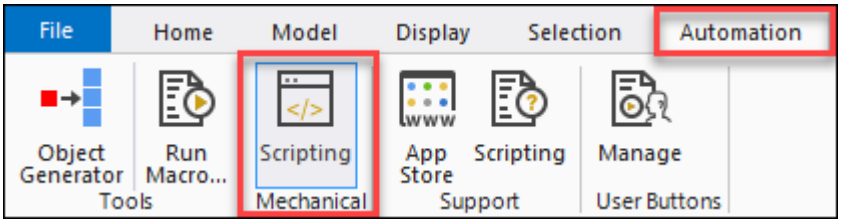
- On the **ACT Start Page**, click the **ACT Console** button in the toolbar.
- Select **Extensions** → **View ACT Console**.

In DesignModeler:

You can open and close the console by clicking the first button in the **ACT Development** toolbar. This toolbar is available only when debug mode is enabled. For more information, see Debug Mode (p. 169).

In Mechanical:

You can open and close the Mechanical **Scripting** view from the ribbon's **Automation** tab. In the **Mechanical** group, clicking **Scripting** switches between opening and closing this view. For more information on using the Mechanical **Scripting** view, see Scripting Introduction in the *Scripting in Mechanical Guide*.



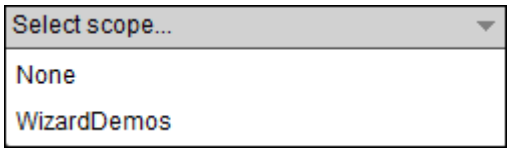
Note:

You can revert to the **ACT Console** by changing the scripting view preference under **File > Options > Mechanical > UI Options > New Scripting UI**. Mechanical must be restarted to see the scripting view change.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Selecting the Scope

Each ACT extension runs with a dedicated script engine. To access global variables or functions associated with an extension, you must first select the extension from the **Select scope** list. This list is populated with the names of all scripted extensions that are loaded for use with the active Ansys product. The following image shows a possible **Select scope** list when only the supplied extension **`WizardDemos`** is loaded.



Entering Commands in the Command Line

In the command line, you can enter commands by typing them, pasting them, or clicking a snippet. When typing a command, you typically begin with **`ExtAPI`**. This stands for *Extension API* and is the gateway into the ACT API. Once you type the period (.), you are in the ACT API.

The console provides smart autocompletion (p. 59), which means that syntax elements such as brackets or quotations marks are taken into account.

You use the following key combinations to insert new lines and execute commands:

Key Combination	Action
Shift + Enter	Insert a new line
Enter	Execute the commands

Additional keyboard shortcuts are described in Using Console Keyboard Shortcuts (p. 67).

Executed commands display as entries in the command history (p. 56).

Working with Command History Entries

When you execute commands, the entries that display in the command history are color-coded by item type. To work with these entries, you use key combinations, icons, and context menu options.

History Color-Coding

In a command history entry, you identify the type of each item by its text color:

Text Color	Item Type
Black	Inputs
Blue	Outputs
Red	Errors

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

History Key Combinations

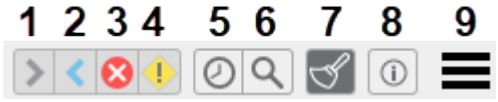
To move between command history entries, you use the following key combinations:

Key Combination	Action
Ctrl + up-arrow	Go to the previous entry
Ctrl + down-arrow	Go to the next entry

Additional keyboard shortcuts are described in Using Console Keyboard Shortcuts (p. 67).

Command History Toolbar Buttons

To work with command history entries, you use these buttons on the toolbar:



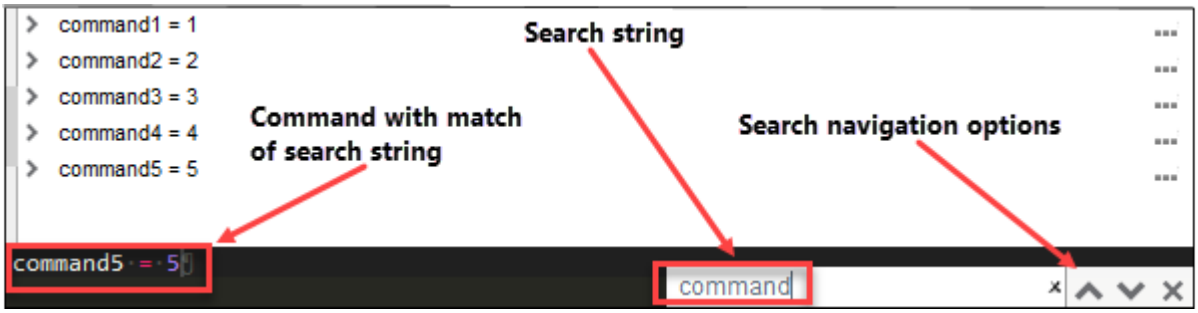
1. Show or hide the history of inputs.
2. Show or hide the history of outputs.
3. Show or hide the history of errors.
4. Show or hide the history of warnings.
5. Display the last executed commands (maximum of 500).
6. Search command history for an entered search string.
7. Clear the command history log.
8. Open the **Help** window.
9. Show hidden icons (visible only when the **ACT Console** window is not wide enough to display all icons in the toolbar).

History Search

When you click the search toolbar button ( ) to search the command history, a reverse search is performed using the text currently entered in the command line. The following figure shows five executed commands and **`command`** entered in the command line.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

To start the search, either click the search toolbar button ( ) or press **Ctrl + R**. A search box appears to the right of the command line. Besides displaying the current search string, the search box provides buttons for stepping backward and forward through the command history entries and a button for closing the search box. When the search starts, it immediately steps backward through the command history, stopping when it finds an entry with a match. In this example, the search stops when it finds **command5 = 5**, displaying this entry in the command line. If you wanted to execute this command again, you would press **Enter**.



To resume the reverse search, you either press **Ctrl + R** or click the up arrow button for the search box. To change the search direction to step forward rather than backward in the history, you either press **Ctrl + Shift + R** or click the down arrow button for the search box. To clear the search string, you click the button with the small X to the right of the search string.

As long as the result in the command line matches the current search string, no change occurs. When the result no longer matches the search string, the search automatically steps backward to find the next match. Assume that you enter **command2** as the search string. Because **command5 = 5** no longer matches **command2**, the search steps back through the command entries to find a match, displaying **command2 = 2** in the command line. Now assume that you enter **command21** as the search string. The search box turns red because no match exists in the command history.

If you enter **command** as the search string once again, the search box is no longer red. Because **command2 = 2** still matches the current search string, it remains displayed in the command line.

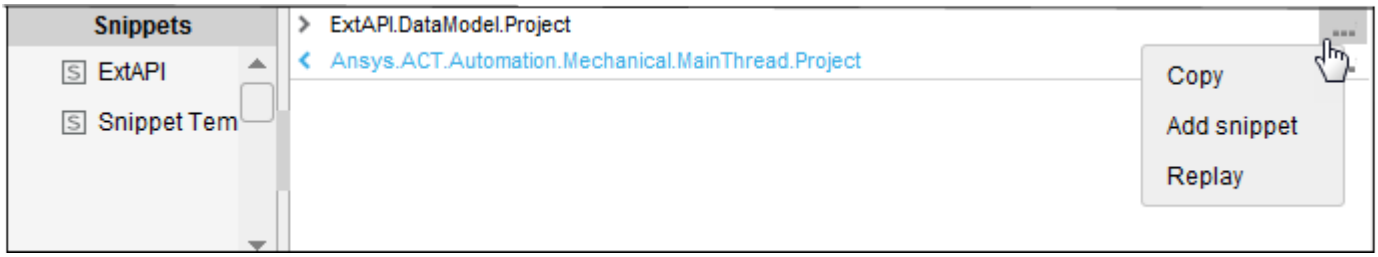
If the search reaches the end of the history without finding a match, it wraps around to the beginning of the command history. When you finish searching, you click the close button for the search box.

History Context Menu

For a command history entry, you can access a drop-down menu and select an action to apply to it. When you click the button with the three dots to the right of an entry, the following actions are available:

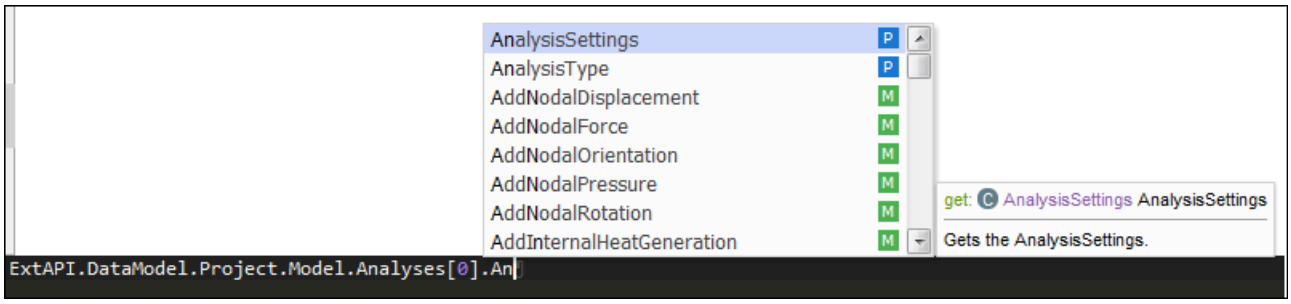
- **Copy**: Copy the command text to the clipboard.
- **Add snippet**: Add the command text to the **Snippets** panel. For more information, see Using Snippets (p. 62).
- **Replay**: Paste the command text into the command line so that it can be executed again later.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Using Autocompletion

As you type in the command line, a scrollable list of suggestions displays above the command line.



Each suggestion has an icon with a letter and color-coding to indicate the type of suggestion. A letter in a square icon describes the type of the member in the suggestion list. A letter in a circle icon describes the nature of the return type.

Listed alphabetically are types of members, which are shown in square icons:

- **E** = Enumeration
- **M** = Method
- **N** = Namespace
- **P** = Property
- **S** = Snippet
- **T** = Type
- **V** = Variable
- **–** = Unidentified member (generally a Python type that cannot be extracted)

Once a member is selected, the nature of the return types available for this member are shown in circle icons. Listed alphabetically are return types:

- **C** = Class
- **E** = Enumeration
- **F** = Field

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

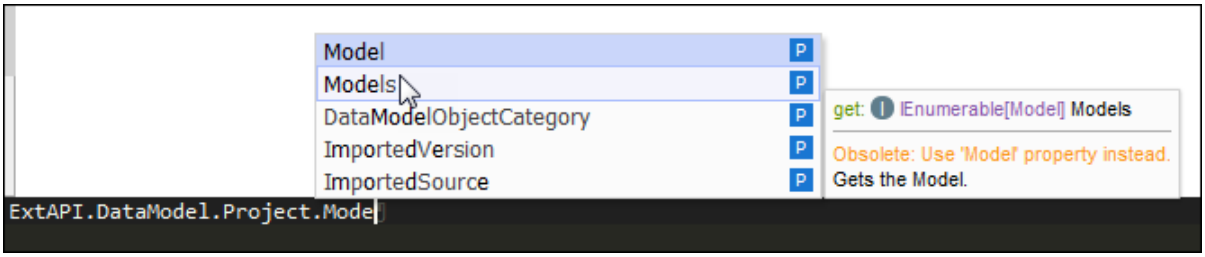
- **I** = Interface
- **S** = Structure

Note:

No icon displays for primitive types (such as integer, Boolean, or string) or if no suggestion is returned.

Using Autocompletion Tooltips

When you place the mouse cursor over any property or method in the list of suggestions, the tooltip displays information about this item. The following image shows the tooltip for the property **`Models`**, which is available when using the console with a project model associated with Ansys Mechanical.



Tooltips use color-coding to indicate the syntax:

- **green** = accessibility
- **purple** = type
- **orange** = warning
- **blue** = arguments

General formatting follows for properties and methods.

Properties:

- **`get/set mode: ReturnType PropertyName`**
- Description of the property.
- **Returns:** Description of what is returned (if any)
- **Remarks:** Additional information (if any)
- **Example:** Sample entry (if any)

Methods:

- **`ReturnType MethodName (ArgumentType ArgumentName)`**
- Description of the method

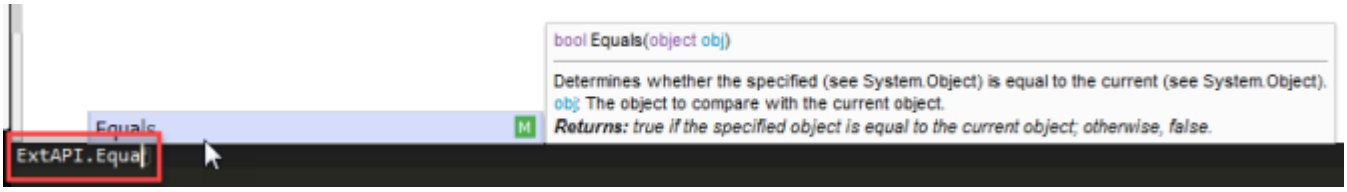
Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

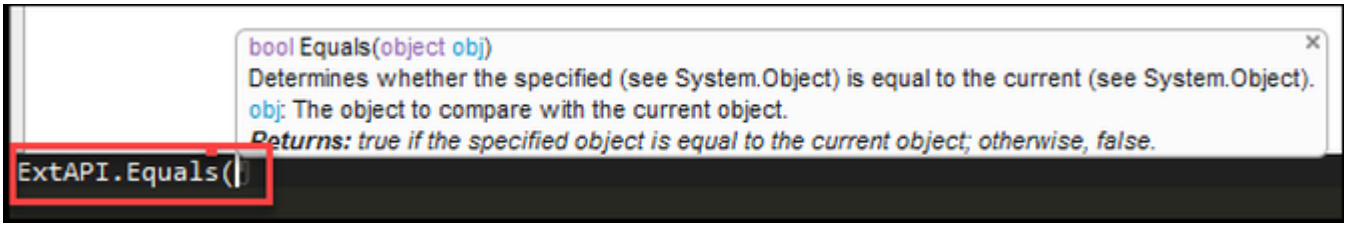
- **`ArgumentName`**: Description of the argument
- **Returns:** Description of what is returned (if any)
- **Remarks:** Additional information (if any)
- **Example:** Sample entry (if any)

Tooltips can also provide:

- **Scope variables** (accessed by pressing **Ctrl + space** when no defined symbol appears under the text cursor).
- **.NET** properties where applicable.
- **Warning messages** when special attention is needed.
- **Prototype information** for methods when cursor is inside brackets and indexers when cursor is inside square brackets.
- **Overloaded methods**, including the following details:
  - **Number** of overloaded methods.
  - **Prototypes** for overloaded methods (accessed by pressing the arrows in the prototype tooltip).
  - **Members** for overloaded methods. (The tooltip for an overloaded method is a list of all members from all overloaded methods.)

The following images show examples of two different tooltips for the method **`Equals`** in two different stages of using autocompletion.





Interacting with Autocompletion Suggestions

You can use the following key combinations to interact with autocompletion suggestions:

Key Combination	Action
Ctrl + space	Open the suggestion tooltip

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Key Combination	Action
Esc	Close the suggestion tooltip
Enter	Apply the suggestion
space or symbol Supported symbols: `. ( ) [ ] { } ‘ “ # , ; : ! / = ?`	Apply the suggestion followed by the space or symbol

For available keyboard shortcuts, see Using Console Keyboard Shortcuts (p. 67).

Using Snippets

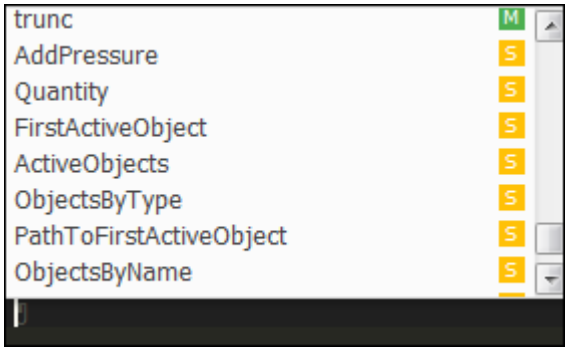
Snippets are existing lines of code that you can quickly and easily insert in the command line, saving you from repetitive typing. As you use the console, you can insert any of the snippets that Ansys supplies for ACT-specific commands. Additionally, you can begin building your own library of snippets to either supplement or replace supplied snippets.

Descriptions follow for the supplied entries in the **Snippets** panel:

- The snippet **ExtAPI** inserts `ExtAPI.` in the command line, providing you with immediate access to the API. From the autocomplete options in the tooltip, you can then begin selecting additional attributes to build your command.
- The snippet **Snippet Template** provides sample code for swapping two variables. The comments explain how a snippet can contain editable fields, which are automatically selected for modification when the snippet is inserted. When a field is selected, you can type a new value to override the default value or let the default value remain. Pressing the **Tab** key moves the cursor to the next editable field.

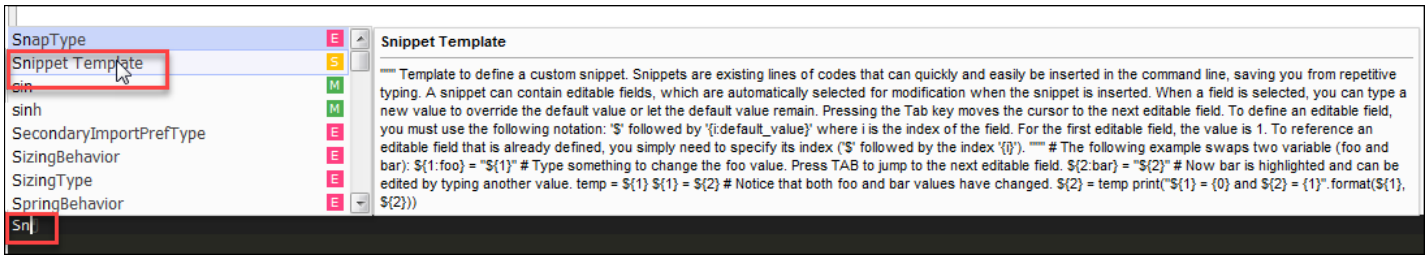
Viewing and Inserting Available Snippets

When nothing is entered in the command line, pressing **Ctrl + space** opens a list of suggestions displaying all available variables and snippets. When you scroll down to the snippets, you can see those predefined for standard Python commands and those that appear in the **Snippets** panel.

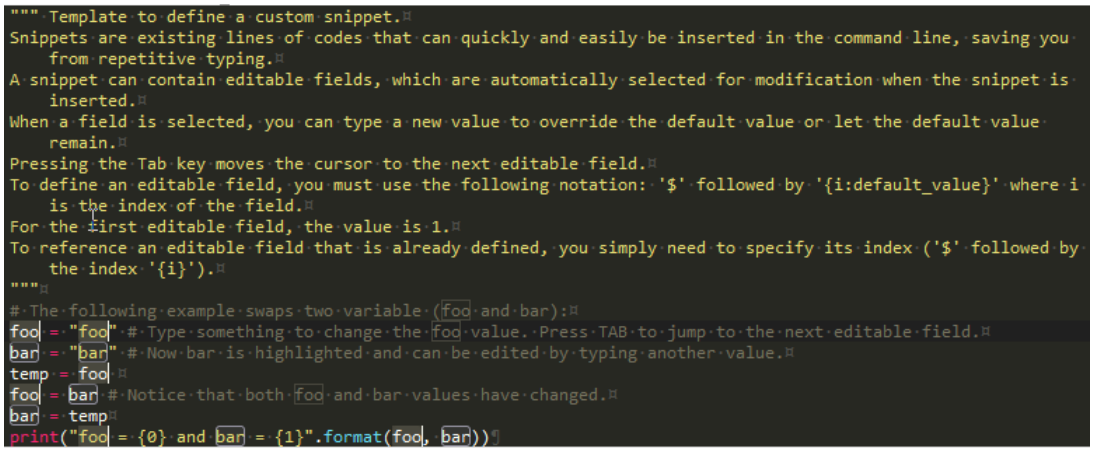


If you know the name of a snippet, you can begin typing the name to find it in the list of suggestions. The supplied **Snippet Template** provides code for swapping two variables.

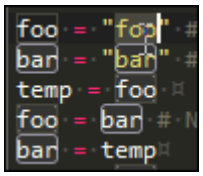
Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



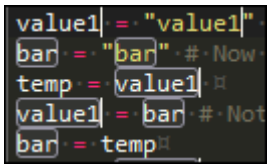
When a snippet selected, you can easily see all editable fields in the command line.



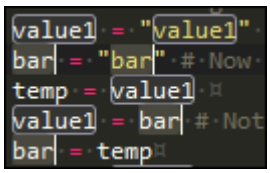
For example, in **Snippet Template**, the first editable field (**`foo`**) is highlighted.



Typing something changes the **`foo`** value to whatever you type (**`value1`**). Notice that both **`foo`** values change to **`value1`** in one operation.

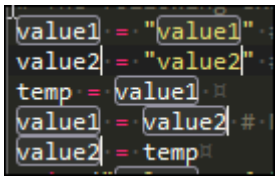


Pressing the **Tab** key moves the cursor to the next editable field, causing **`bar`** to be highlighted.



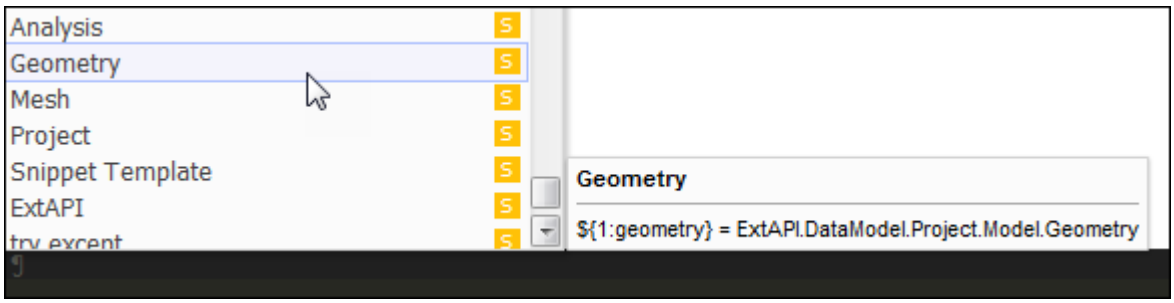
Typing something changes the **`bar`** value to whatever you type (**`value2`**). Notice that both **`bar`** values change to **`value2`** in one operation.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Pressing the **Tab** key again finalizes the code.

To define an editable field, you must use the following notation: **${#:#}**, where # is the index of the field. For the first editable field, the value is **1**. To reference an editable field that is already defined, you simply need to specify its index (**${#}**) as shown in the following figure for the snippet **Geometry**.



For the previous example for swapping two variables, you can define the following snippet:

```
temp = ${1:foo}
${1} = ${2:bar}
${2} = temp
```

Creating and Managing Snippets

The **Snippets** panel displays all personal snippets, which are either those supplied by Ansys for ACT-specific commands or those that you create. It does not display snippets for basic Python commands because these snippets cannot be removed from the console. For example, the **Snippets** panel displays the supplied **Snippet Template** for swapping two variables, but it does not display the snippet **lambda** for this basic Python command.

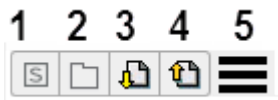
In the **Snippets** panel, you can click a snippet to insert it in the command line.

Note:

When typing in the command line, autocompletion displays all snippets in the **Snippet** panel as well as snippets for basic Python commands.

To make creating and managing snippets easy, the console panel provides toolbar bars and a context menu.

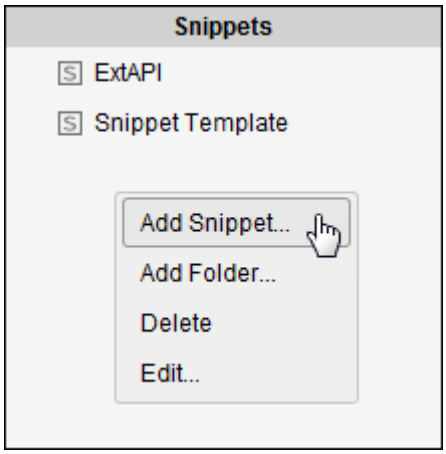
Snippet Toolbar Buttons



Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

1. Add a new snippet.
2. Add a new snippet folder.
3. Import an existing snippets collection.
4. Export the current snippets collection.
5. Show hidden icons (visible only when the console window is not wide enough to display all icons in the toolbar).

Context Menu



Creating Snippets and Folders

You can create snippets and folders using any of the following methods:

- Click the button with the three dots to the right of an entry in the command history and select **Add snippet**.
- Click the toolbar button for adding a new snippet or folder.
- Right-click a snippet or folder in the **Snippets** panel and select **Add Snippet** or **Add Folder**. Where you right-click determines where the new snippet or new folder is placed in the **Snippets** panel.
  - When you right-click a folder, the new item is created inside this folder.
  - When you right-click a snippet, the new item is created at the bottom of the containing folder.

Once the snippet or folder is created, the **Snippet Properties** dialog box opens so that you can specify properties. A snippet has two properties, **Caption** and **Value**. A folder has only **Caption**.

- **Caption** specifies the text to display as the name for the snippet or folder.
- **Value** specifies the code that the snippet is to insert in the command line. When a snippet is created from an entry in the command history, **Value** displays the code for this entry by default.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

After specifying the properties for the snippet or folder, you click **Apply** to save your changes or **Cancel** to discard the newly created snippet or folder.

Viewing and Editing Snippet or Folder Properties

To view or edit the properties of an existing snippet or folder in the **Snippets** panel, right-click it and selecting **Edit** to open the **Snippet Properties** dialog box. When finished, click **Apply** to save changes or **Cancel** to exit without saving changes.

Deleting Snippets and Folders

To delete a snippet or folder from the **Snippets** panel, right-click it and select **Delete**.

Caution:

If you delete a folder, all of its contents are also deleted. Deleting a snippet from the **Snippets** panel also removes it from the list of suggestions for autocompletion.

Organizing Snippets and Folders

When creating a snippet or folder, you can determine its location by using a particular creation method, as described in Creating Snippets and Folders (p. 65). To organize existing snippets and folders in the **Snippets** panel, you can drag them to the desired location in the hierarchical tree structure. Using folders, you can organize your snippets to create personal, reusable libraries.

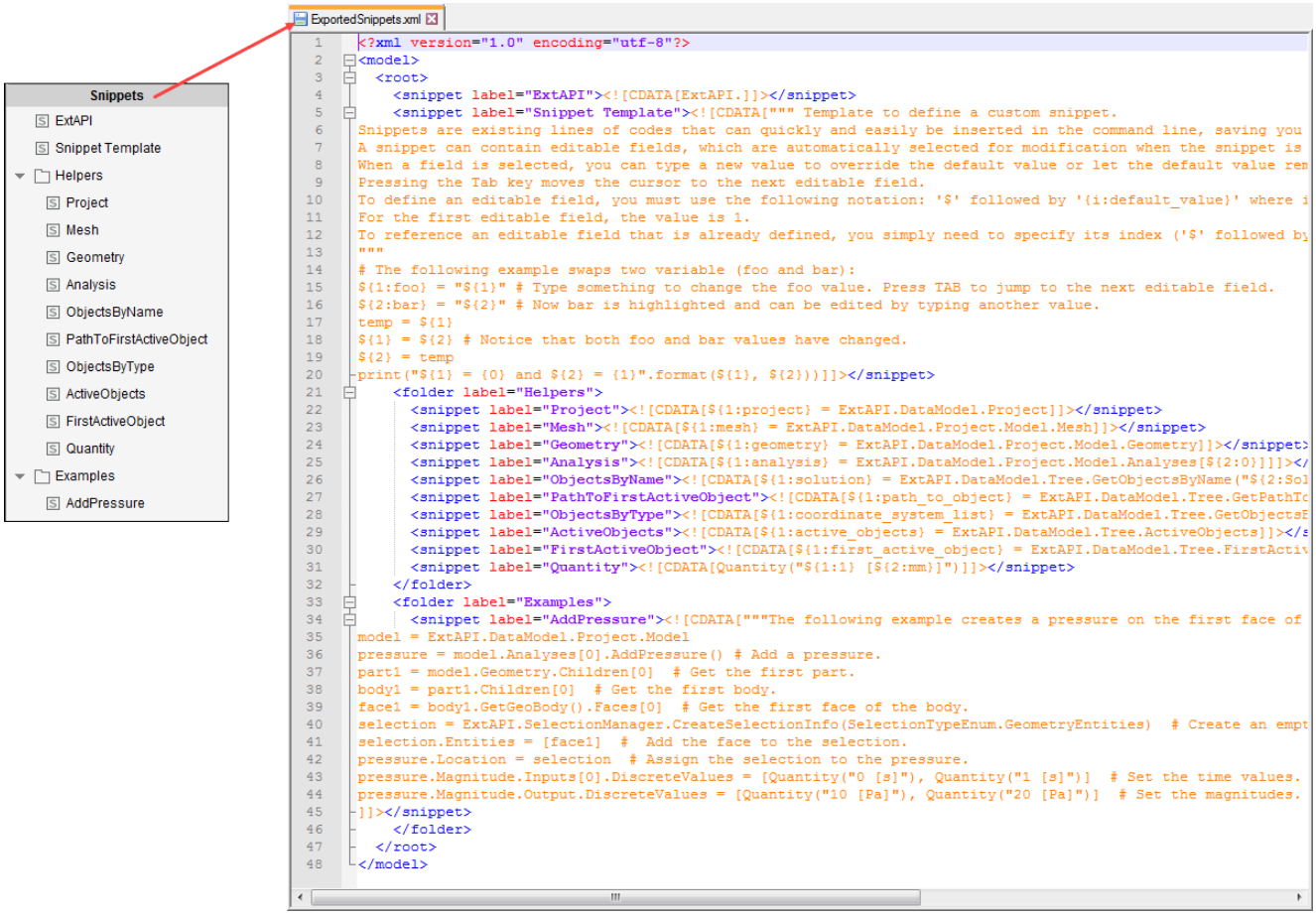
Importing and Exporting a Snippets Collection

You can import or export a collection of snippets as an XML file.

- Clicking the toolbar button for importing a snippets collection () appends the snippets in the selected XML file to the list already shown in the **Snippets** panel.
- Clicking the toolbar button for exporting a snippets collection () exports all snippets in the **Snippets** panel to the specified XML file.

The following figure shows an XML file created from the export of a snippets collection.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Using Console Keyboard Shortcuts

The following tables list keyboard shortcuts for the **ACT Console**:

Line Operation Shortcuts

Key Combination	Action
Ctrl + D	Remove line
Alt + Shift + ↓	Copy lines down
Alt + Shift + ↑	Copy lines up
Alt + ↓	Move lines down
Alt + ↑	Move lines up

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Key Combination	Action
Alt + Backspace	Remove to line end
Alt + Delete	Remove to line start
Ctrl + Delete	Remove word left
Ctrl + Backspace	Remove word right

Selection Shortcuts

Key Combination	Action
Ctrl + A	Select all
Shift + ←	Select left
Shift + →	Select right
Ctrl + Shift + ←	Select word left
Ctrl + Shift + →	Select word right
Shift + Home	Select line start
Shift + End	Select line end
Alt + Shift + →	Select to line end
Alt + Shift + ←	Select to line start
Shift + ↑	Select up
Shift + ↓	Select down
Shift + Page Up	Select page up
Shift + Page Down	Select page down
Ctrl + Shift + Home	Select to start
Ctrl + Shift + End	Select to end
Ctrl + Shift + D	Duplicate selection
Ctrl + Shift + P	Select to matching bracket

Multi-Cursor Shortcuts

Key Combination	Action
Ctrl + Alt + ↑	Add multi-cursor above
Ctrl + Alt + ↓	Add multi-cursor below
Ctrl + Alt + →	Add next occurrence to multi-selection
Ctrl + Alt + ←	Add previous occurrence to multi-selection

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Key Combination	Action
Ctrl + Alt + Shift + ↑	Move multi-cursor from current line to the line above
Ctrl + Alt + Shift + ↓	Move multi-cursor from current line to the line below
Ctrl + Alt + Shift + →	Remove current occurrence from multi-selection and move to next
Ctrl + Alt + Shift + ←	Remove current occurrence from multi-selection and move to previous
Ctrl + Shift + L	Select all from multi-selection

Go-To Shortcuts

Key Combination	Action
Page Up	Go to page up
Page Down	Go to page down
Ctrl + Home	Go to start
Ctrl + End	Go to end
Ctrl + L	Go to line
Ctrl + P	Go to matching bracket

Folding Shortcuts

Key Combination	Action
Alt + L, Ctrl + F1	Fold selection
Alt + Shift + L, Ctrl + Shift + F1	Unfold

Other Shortcuts

Key Combination	Action
Tab	Indent
Shift + Tab	Outdent
Ctrl + Z	Undo
Ctrl + Shift + Y, Ctrl + Y	Redo
Ctrl + T	Transpose letters
Ctrl + Shift + U	Change to lower-case
Ctrl + U	Change to upper-case

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Key Combination	Action
Insert	Overwrite
Ctrl + R	Search command history for match, stepping backward through commands
Ctrl + Shift + R	Search command history for match, stepping forward through commands

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extensions

Extension development consists of creating the scripted extension. During development, enabling debug mode (p. 169) is recommended.

Once you finish developing and debugging a scripted extension, you can use the binary extension builder (p. 49) to compile into a binary extension for sharing with others. You cannot view or edit the contents of a binary extension.

Extensions cannot be executed until you use the Extension Manager (p. 41) to install and load them. For information on installing and loading both scripted extensions and binary extensions, see Installing and Uninstalling Extensions (p. 44) and Loading and Unloading Extensions (p. 46).

The following topics provide extension creation information:

Extension Definition

Extension Configuration

Extension and Template Examples

Ansys Store Extensions

ACT App Builder

Libraries and Advanced Programming

Note:

- An ACT license is required to create scripted extensions and build binary extensions. However, an ACT license is not required to execute binary extensions. For more information, see Licensing (p. 12).
- ACT can manage an entire set of extensions for different ANSY products. When an extension targets multiple products, it loads the features applicable to the running product. If the end user engages another target product, only customizations specific to that product are available.
- Once you save a project in which one or more extensions is loaded, any further use of the project requires the same extensions to be available. If the extensions are found, Workbench loads them automatically when the project is opened. If the extensions are not found, an error message displays. For more information about controlling the available of extensions, see Extension Configuration (p. 78).
- When importing a project into an active Workbench session, ACT merges attribute data from saved extensions. However, the project import operation will fail if same-named attributes exist in identical extensions from both the incoming and active projects. Workbench will report the collision error, abandon the import, and restore the original active project.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extension Definition

An ACT scripted extension has two basic components:

- **XML extension definition file**: Defines and configures the content of the extension using XML elements (p. 185). Each element has a set of required and optional attributes. The element **`<callbacks>`** specifies the functions or methods that the extension is to invoke from all those defined in the extension's IronPython script. While you can create the XML file manually, the ACT App Builder (p. 94) automates its creation.
- **IronPython script**: Defines the functions invoked by user interactions and implements the extension's behavior. The content of the script is not case-sensitive. You can debug the script by entering commands in the ACT Console (p. 53) or taking advantage of the ACT Debugger (p. 170).

Note:

You use the IronPython language to develop the functions that the extension executes. Comprehensive information about IronPython is available at IronPython.net, including documentation. Additionally, obtaining a copy of David Beazley's book, *Python Essential Reference*, is highly recommended. This book is considered the definitive reference guide on the Python language.

For script examples, you have supplied extensions and templates (p. 81) that you can download. Additionally, you can log into the Ansys customer site and search solutions using the following filter selections:

- For **Product**, select **ACT Customization Suite**.
- For **Product Family**, select **Scripting**.

To further limit the solutions shown, you can use the search box at the top of the page.

In addition to the XML file and at least one IronPython script, an extension can include supporting files, such as image files, custom wizard help files (p. 160), input files, and installed IronPython function libraries and C# assemblies (p. 114).

While the content in the XML file and the script depend on the Ansys product that is being customized, the need for these two basic components remains consistent.

The following topics describe how to create a very simple extension named **`ExtSample1`**:

Creating the Extension Definition File

Creating the IronPython Script

Setting Up the Directory Structure

Viewing Exposed Custom Functionality

Creating the Extension Definition File

The supplied extension **`ExtSample1`** customizes Mechanical by adding a toolbar with a single button toolbar. When this button is clicked, a dialog box opens, displaying the following message: **`High`**

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

`five! ExtSample1 is a success!` For extension download information, see Extension and Template Examples (p. 81).

The file `ExtSample1.xml` defines and configures the extension.

```
extension version="1" name="ExtSample1">
  <guid shortid="ExtSample1">2cc739d5-9011-400f-ab31-a59e36e5c595</guid>
  <script src="sample1.py" />
  <interface context="Mechanical">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
    </callbacks>
    <toolbar name="ExtSample1" caption="ExtSample1">
      <entry name="HighFive" icon="hand">
        <callbacks>
          <onclick>HighFiveOut</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
</extension>
```

The XML file for an extension always begins with the element **`<extension>`**. All other elements are children to this base tag or root node. For descriptions of basic elements, see Appendix A: Extension Elements (p. 185). For descriptions of all elements, see the *Ansys ACT XML Reference Guide*.

In this extension, the element **`<entry>`** defines the single toolbar button. The file `hand.bmp` provides the image to use as the icon. The callback **`<onclick>`** defines the name of the function to invoke when the button is clicked. The next section addresses the script defined for the function **`HighFiveOut`**.

Note:

- For images to display as toolbar buttons or next to menu commands, Mechanical requires BMP files. Workbench and all other Ansys products require PNG files.
- To designate a transparent background for a BMP file, in an image editor, set the background color to 192, 192, 192 and then save the file with 256 colors.

Creating the IronPython Script

The IronPython script defines functions that respond to user and interface interactions and implements the behavior of the extensions. Typically, functions are invoked through the different events or callbacks in the XML file.

The script **`sample1.py`** for the extension **`ExtSample1`** follows:

```
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *

def init(context):
    ExtAPI.Log.WriteMessage("Init ExtSample1...")

def HighFiveOut(analysis_obj):
    MessageBox.Show("High five! ExtSample1 is a success!")
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

This script contains two functions:

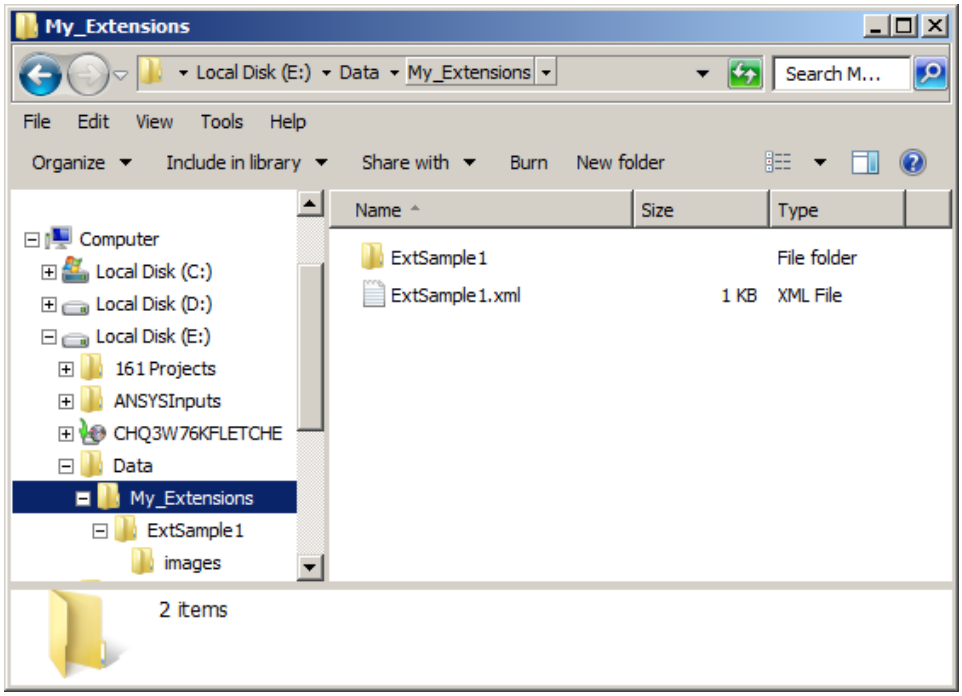
- **`init()`**: Called when the Ansys product is opened. In the XML file, the argument **`context`** for the element **`<interface>`** contains the name of the product (**`"Mechanical"`**).
- **`HighFiveOut()`**: Called when the toolbar button **HighFive** is clicked. The callback **`<onclick>`** passes an instance of the active object **`Analysis`** as an argument.

For any function, the global variable **`ExtAPI`** represents the main entry point to the ACT API (p. 53). In this example, the function **`init()`** uses the ACT interface **`ILog`** to write a message in the log file. For comprehensive information on available interfaces, see the Ansys ACT API Reference Guide.

Setting Up the Directory Structure

In the directory where you store your extensions, the XML file and the folder with the extension's script and supporting files are stored at the same level. Both the XML file and the folder must have the same name as the extension.

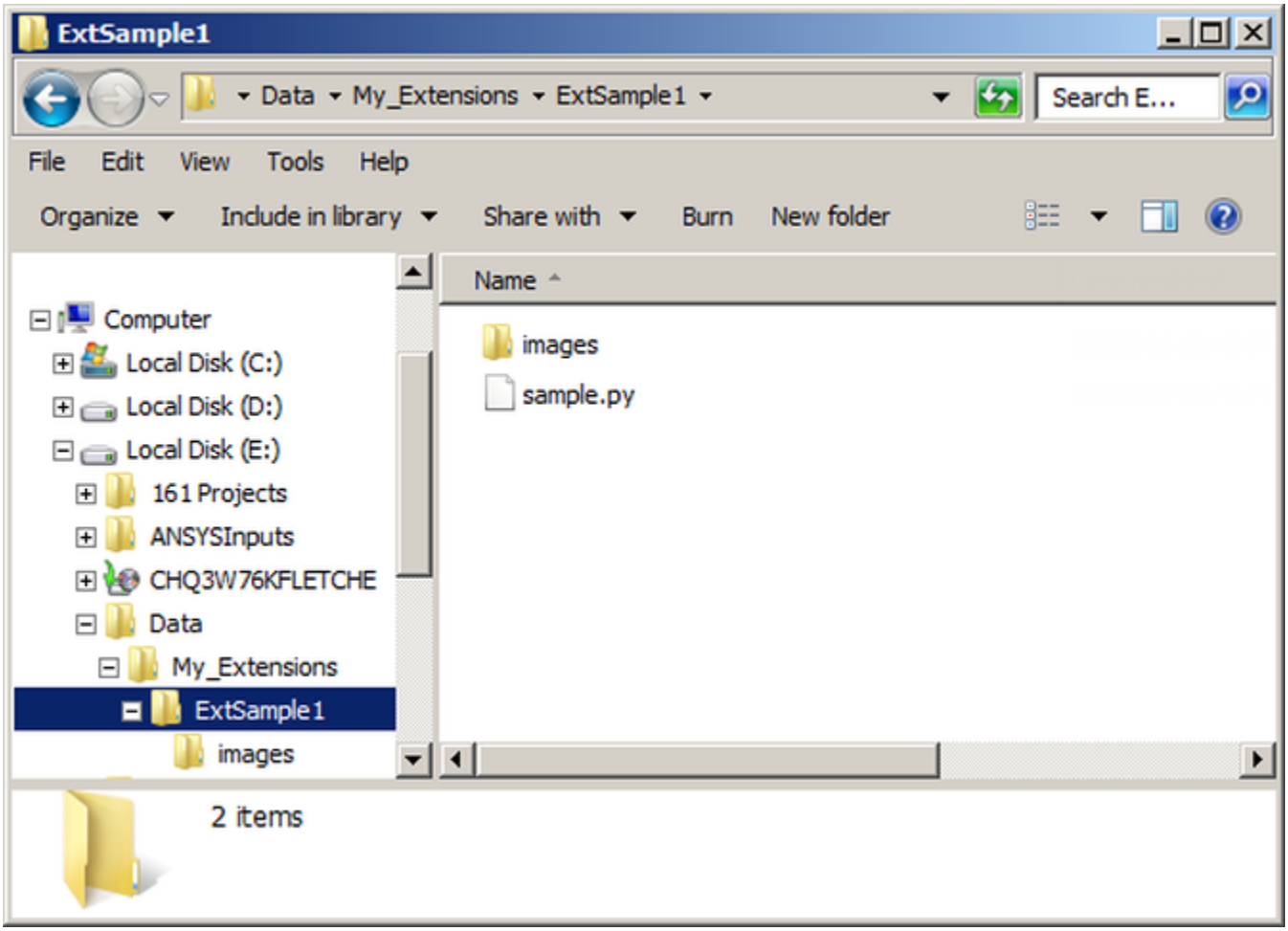
The following figure shows the top-level directory structure for the extension **`ExtSample1`**. You can see that the extension's XML file and its folder are both named `ExtSample1`.



The following figure shows the contents of the folder `ExtSample1`.

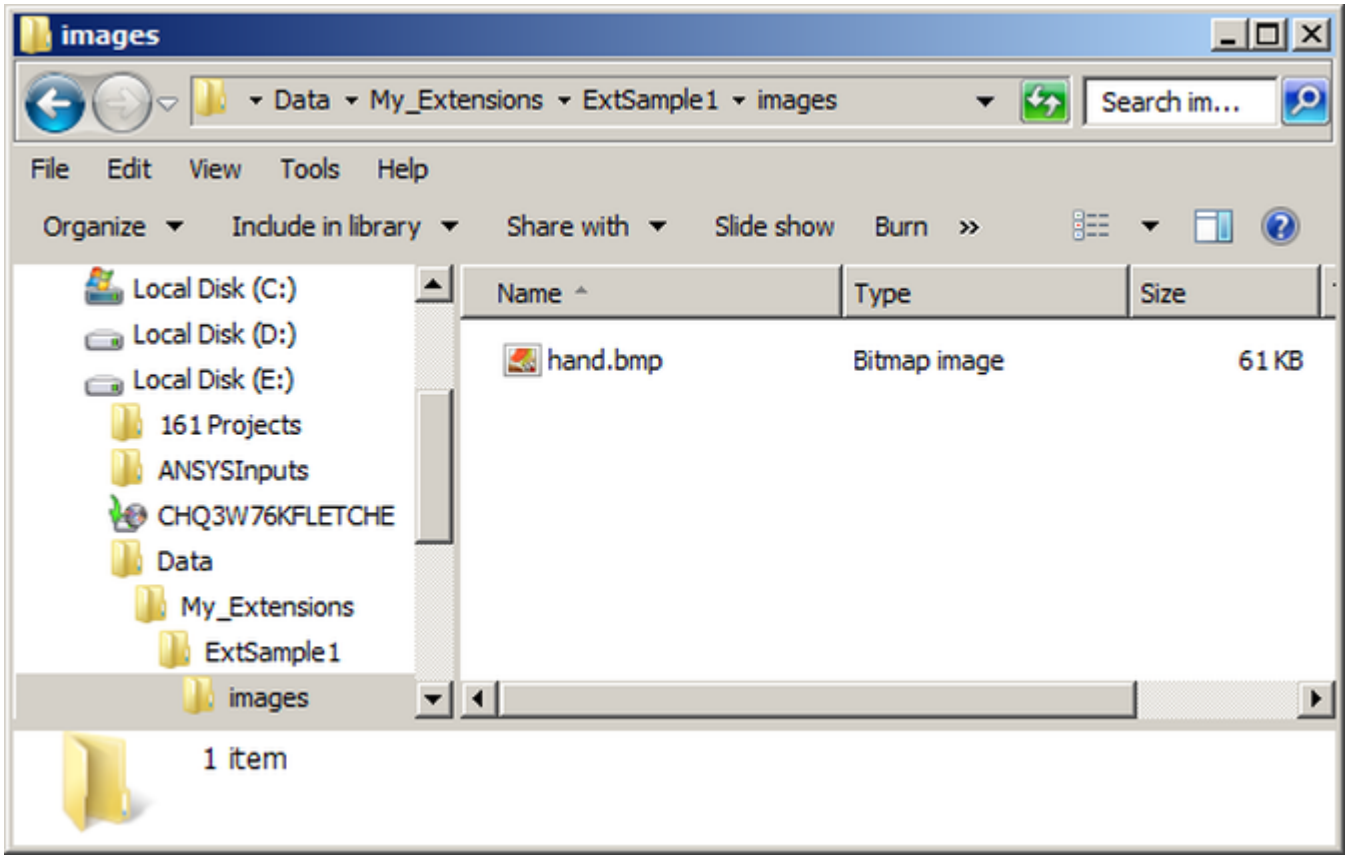
- The script `sample1.py` contains the code that implements the extension's behavior. By default, ACT looks for the script in the directory of the extension. If the script is located in a different directory, in the XML file, the element **`<script>`** must specify an explicit path to the script.
- The folder **`images`** contains the one or more image files to display as icons in the interface of this extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



The following figure shows the contents of the folder **`images`**. The image file `hand.bmp` displays as the icon for the custom button that is exposed on the tab for this extension when it is loaded in Mechanical.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Custom Menu Icons in the Mechanical Ribbon User Interface

Introduced in 2019 R2, the Mechanical ribbon user interface handles menu entry icons differently than earlier Mechanical versions. To achieve optimum image compatibility:

1. Create four Portable Network Graphics (PNG) files of your icon in these specific pixel sizes: 16x16, 24x24, 32x32, and 48x48.

2. Apply this naming convention:

   `icon_`*`yourIconName`*`_`*`sizeAxsizeA`*`.png`

   For example, assume that you specify **`"myIcon"`** as the icon attribute value in your extension XML file:

   ```
   <entry name="myEntry" icon="myIcon" ... />
   ```

   You must then name your icon files as follows:

   - **`icon_myIcon_16x16.png`**
   - **`icon_myIcon_24x24.png`**
   - **`icon_myIcon_32x32.png`**
   - **`icon_myIcon_48x48.png`**

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

3. Store all four icon files in your extension's original `images` folder.

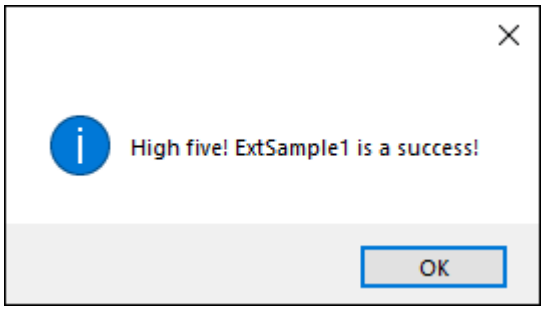
Note:

- You can use your preferred image manipulation tool to create the four PNG files in the required pixel sizes. Using the functionality for scaling an image, start with the maximum resolution (64x64 pixels) and work down to the minimum resolution (16x16 pixels), exporting a new image after each re-scale operation.
- Tree object icons continue to use the bitmap image format in the 16x16 pixel size. You do not need to update these icons for existing extensions. In the XML file for a new extension, you specify the entire name (without the **`.bmp`** extension), just as you did before the ribbon UI change.

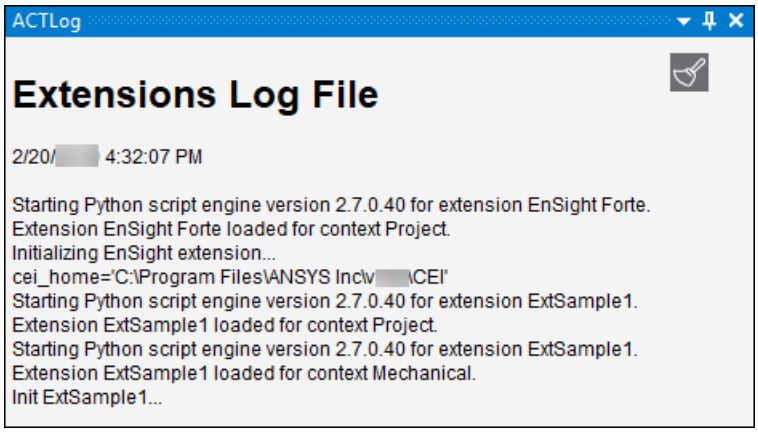
Viewing Exposed Custom Functionality

In the extension **`ExtSample1`**, the attribute **`context`** is set to **`Mechanical`**. When this extension is loaded in Mechanical, the ribbon displays the **ExtSample1** tab to the right of the **Automation** tab.

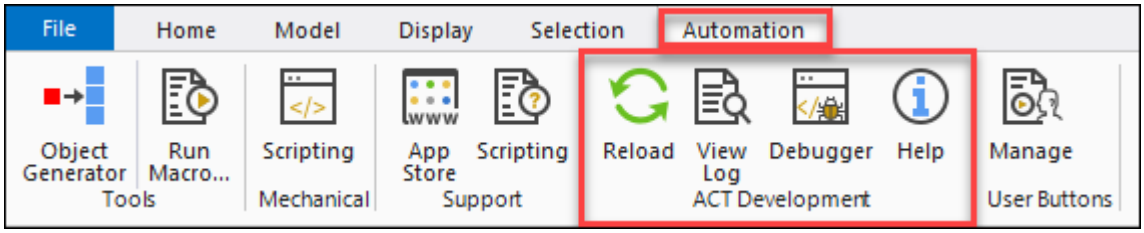
Clicking the **High Five** button on this tab executes the extension, which opens this dialog box:



The following figure shows the **Extensions Log File**. It indicates that the extension **`ExtSample1`** is loaded for the context **`Project`**.



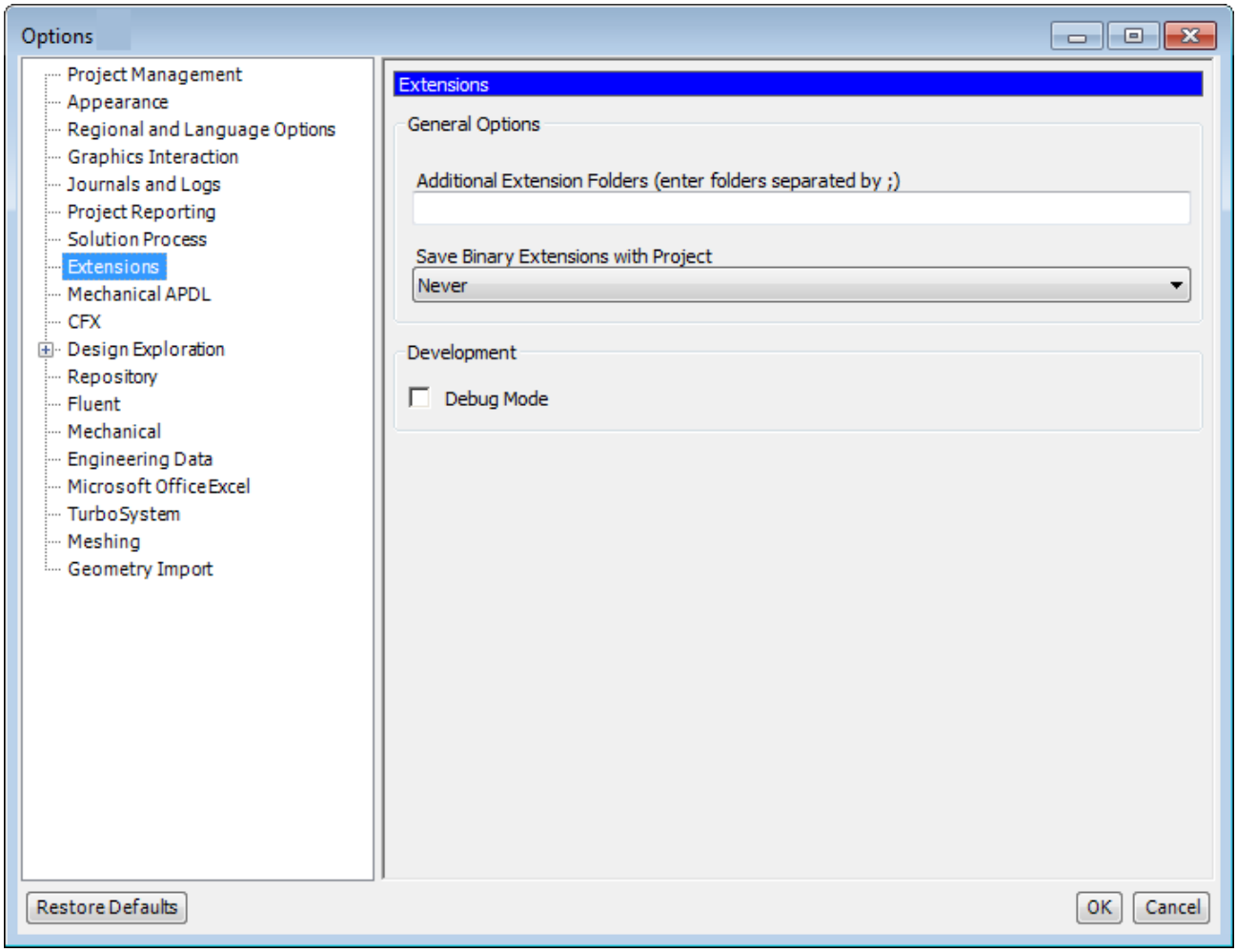
In Mechanical, if debug mode is enabled, you can open this file from the **ACT Development** group on the **Automation** tab. Clicking **View Log** switches between opening and closing the **Extensions Log File**.



Extension Configuration

You select **Tools** → **Options** → **Extensions** to configure ACT extension options.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Descriptions of these options follow:

Additional Extension Folders Option

Save Binary Extensions with Project Option

Debug Mode Option

Additional Extension Folders Option

The **Additional Extension Folders** option defines the additional folders in which ACT searches for the extensions that the Extension Manager (p. 41) is to display. You can define several folder names, separating them with a semicolon (;).

Folders that you define here are added to your **`ApplicationData`** folder: **`%APPDATA%\Ansys\v222\ACT\extensions`**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Because these folders are searched for extensions by default, the Extension Manager (p. 41) displays any extensions located in them.

Note:

The default folders are not searched recursively. Because extensions stored in subdirectories are not found, store your extensions at the top level of the directory.

During this process, warning messages are returned for any conflicts. These messages are logged in the Extensions Log File (p. 47).

Note:

The **Additional Extension Folders** option is also available when you click the gear icon in the graphic-based Extension Manager (p. 41).

Save Binary Extensions with Project Option

The **Save Binary Extensions with Project** option specifies whether to save binary extensions within the project when the project is saved. Choices are:

- **Never** (default): The currently loaded extensions are not saved within the project.
- **Copied but locked to the project**: Extensions are saved within the project but are limited to only this project.
- **Always**: The extensions are saved within the project and no restrictions exist as to their use in other projects. This option represents what is generally expected when the project is saved. However, the behavior depends on the security option that was selected when the binary extension was built. For more information, see Binary Extension Builder (p. 49). In particular, the following scenarios can occur:
  - The extension was built with the security option set to **Can't copy extension to project**. If the save option is set to **Always** or **Copied but locked to the project**, the security option has the priority. The extension is not saved within the project.
  - The extension was built with the security option set to **Can copy extension to project but locked to this one**. If the save option is set to **Always**, it does not impose any restriction on the extension. However, the security level limits the use of the extension to only the given project.

Caution:

Saving extensions with a project imposes specific project load behavior for anyone opening the project file. When opening a Workbench project (.wbpj) or archive (.wbpz) that includes saved extensions, the **Extension Manager** will uninstall the local versions and remove the local extension files from disk. The project will only load and use the saved extensions. The safest project option, and default value, is **Never**. Otherwise, a user may unintentionally delete installed extensions from %APPDATA%, `Addins\ACT\extensions`, or any additional extension directory set by preference.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Debug Mode Option

The **Debug Mode** option enables and disables debug mode for the debugging of IronPython scripts for loaded extensions. When this check box is selected, debug mode is enabled. For more information about debug mode and the additional features that it exposes, see Debug Mode (p. 169).

Note:

The **Debug Mode** option is also available when you click the gear icon in the graphic-based Extension Manager (p. 41).

Extension and Template Examples

The following topics provide descriptions of supplied extension and template examples. You can use these examples to learn how to use ACT:

Supplied Extensions

Supplied Templates

Extension Versus Template

All supplied examples are extensions, regardless of their package labels. The package **ACT Extension Examples** provides extensions that ACT documentation specifically references and explains. Template packages provide additional extensions that you can expand and manipulate to create your own custom extensions. While a few of the extensions in the template packages are referenced and explained in ACT documentation, most are not. You can explore these additional extensions as they become applicable to your own extension creation objectives.

Download Information

You can easily download the extension and template packages from the help panel for the ACT Start Page (p. 39):

- To download the package of extension examples, click **Extension Examples** under **Downloadable Examples**.
- To download template packages, click the links for the packages that you want to download under **Downloadable Examples**.

Note:

On the Ansys Store, the App Developer Resources page provides for downloading these packages from links on the **Help & Support** and **ACT Templates** tabs.

After unzipping a package, you can install and load the extensions that they contain if you have an ACT license. For more information, see Installing and Uninstalling Scripted Extensions (p. 44).

If you want, you can import any existing extension into the ACT App Builder (p. 94) to see and edit its files in a visual environment.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Supplied Extensions

The package of extension examples contains the many extensions that the ACT documentation references. This table describes these extensions and provides links to where they are explained in the ACT documentation. For an extension specific to an Ansys product, the referenced topic is in the ACT customization guide for the particular product. For download information, see the previous topic.

Extension	Product Context	Description	Reference
ACTResults	Mechanical	Shows how to add postprocessing features in Mechanical. This extension adds a toolbar and buttons that create ACT objects in the Mechanical tree to access a result for a shell element, layer element, and contact and then display them as custom results. The creation of each result is explained in its own topic.	• Accessing Results on Shell and Layer Elements • Creating a Custom Stress Result on a Shell Element • Creating a Custom Stress Result on a Layer Element • Creating a Custom Result on a Contact
AdvancedProperties	Mechanical, but applicable to others	Adds a property group and a time-dependent worksheet (property table) to Mechanical.	Dialog Box Creation (p. 132)
Coupling	Mechanical	Shows how to create a tool for coupling two sets of nodes related to two edges. This Mechanical extension demonstrates how you can develop your own preprocessing feature, such as a custom load, to address one specific need.	Coupling Two Sets of Nodes
CustomLayout	Workbench, but applicable to others	Shows how to create the extension **`CustomLayoutWizard`**, which has customized wizard-level and component-level layouts.	Custom Wizard Interface Example (p. 164)

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extension	Product Context	Description	Reference
CustomTransfer	Workbench	Shows how to implement a custom transfer from a producing task group to a consuming task group, creating connections between custom tasks. This Workbench extension also illustrates the creation of single-task vs. multiple-task group.	Custom Transfer
DataSquares	Workbench	Shows how to use custom task properties to integrate an external application and submit a task to the Ansys Remote Solve Manager. This Workbench extension uses custom task properties to drive an external application in squaring an input number. The result is displayed in the **Parameter Set** bar in Workbench.	External Application Integration with Custom Data and Remote Job Execution
DebuggerDemo	Workbench, but applicable to others	Shows how to use the **ACT Debugger** to debug the IronPython scripts for loaded extensions.	ACT Debugger (p. 170)
DemoLoad	Mechanical	Shows how to add a preprocessing feature for Mechanical. This extension adds a toolbar and buttons to create a generic load.	Adding a Custom Load
DemonstrationSolver	Mechanical	Shows how to add a connection to a third-party solver in Mechanical. This extension provides the ability to start an external solver rather than an Ansys solver.	This extension is included in the package but is not referenced in the documentation.
DemoResult	Mechanical	Shows how to add a postprocessing feature in Mechanical. This extension adds a toolbar and buttons to create a custom result. The custom result computes the worst value of the principal stresses for the scoped geometry entities.	This extension is included in the package but is not referenced in the documentation.
EmptyGUI	Workbench	Shows how to add a custom context menu for a task to the Workbench **Project Schematic**.	Custom User-Specified Interface Operation

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extension	Product Context	Description	Reference
ExtDialogSample	Mechanical, but applicable to others	Shows how to add a dialog box.	Dialog Box Creation (p. 132)
ExtSample1	Mechanical, but applicable to others	Shows how to add a toolbar with a single button.	Extension Definition (p. 72)
ExtSolver1	Mechanical	Shows how to connect to a very simple external solver in Mechanical.	Third-Party Solver Connections in Mechanical
ExtToolbarSample	Mechanical, but applicable to others	Shows how to add a toolbar with three buttons.	Toolbar Creation (p. 127)
GenericMaterialTransfer	Workbench	Shows how to implement two custom material transfer task groups, with each passing material data to a downstream task group or task.	Generic Material Transfer
GenericMeshTransfer	Workbench	Shows how to implement two custom mesh transfer task groups, with each including *consuming* and *providing* connections.	Generic Mesh Transfer
Mises	Mechanical	Shows how to define a custom result in Mechanical. This custom result reproduces the result for the averaged von Mises equivalent stress.	This extension is included in the package but is not referenced in the documentation. For similar examples, see the custom stress examples in Postprocessing Capabilities in Mechanical.
Parametric	Workbench	Shows how to create a parametric task group in Workbench using the attribute **`isparametricgroup`**.	Parametric Task Group
SC_BGAExtension	SpaceClaim	Shows how to create a wizard named **`BGAWizard`** that is executed from SpaceClaim. This wizard shows how to generate a ball grid array (BGA), which is a surface on which to mount integrated circuits.	Space Claim Wizard for Generating a Ball Grid Assembly (BGA)

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extension	Product Context	Description	Reference
Squares	Workbench	Shows how to use parameter definitions to integrate an external application in Workbench. This example uses parameters to drive an external application in squaring an input number. The result is displayed in the **Parameter Set** bar in Workbench.	External Application Integration with Parameter Definition
TraverseExtension	Mechanical	Shows how to traverse the nodes in the Mechanical tree. This example is described in the *Scripting in Mechanical Guide*. It shows how to use Mechanical APIs to query properties for the objects **`Geometry`**, **`Mesh`**, **`Simulation`**, and **`Results`**.	Object Traversal
WizardDemo	Electronics Desktop	Shows how to create a wizard named **`ED Wizard Demo`** to execute from Electronics Desktop.	Electronics Desktop Wizards
WizardDemos	Multiple Ansys products	Shows how to create multiple wizards running across multiple Ansys products in a single extension.	
	BridgeSimulation	Shows how to create a mixed wizard named **`BridgeSimulation`** that runs wizards from the **Project** tab in Workbench and from DesignModeler or SpaceClaim, and Mechanical.	Mixed Wizard Example (p. 155)
	Workbench	Shows how to create a wizard named **`ProjectWizard`** to execute from the **Project** tab in Workbench.	Wizard Creation (p. 144)
	DesignModeler	Shows how to create a wizard named **`CreateBridge`** to execute from DesignModeler.	DesignModeler Wizards
	SpaceClaim	Shows how to create a wizard named **`CreateBridge`** to execute from SpaceClaim.	SpaceClaim Wizard for Building a Bridge
	Mechanical	Shows how to create a wizard named **`SimpleAnalysis`** to execute from Mechanical.	Mechanical Wizards
WorkflowCallbacksDemo	Workbench **Project** tab	Shows how to use global callbacks to implement a custom process or action before	Global Workflow Callbacks

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Extension	Product Context	Description	Reference
		or after a Workbench **Project Schematic** operation. This extension example adds pre- and post-operation callbacks for each available operation.	

Supplied Templates

Descriptions follow for the extensions in supplied template packages:

ACT Templates for DesignModeler

ACT Templates for DesignXplorer

ACT Templates for Mechanical

ACT Wizard Templates

ACT Workflow Templates

For download information, see Extension and Template Examples (p. 81).

ACT Templates for DesignModeler

The following table describes the folders in the template package for DesignModeler. Each folder contains an extension example. For information about using ACT to customize DesignModeler, see the *ACT Customization Guide for DesignModeler*.

Folder	Description
DM_Template1-SelectionManager	Shows how to add toolbar items to DesignModeler, access model entities, generate criteria-based entity listings, and interact with the **Selection Manager** by adding entities to it.
DM_Template2-PropertyGroup	Shows how to add toolbar items to DesignModeler, add custom objects to the tree, add properties and property groups to a custom tree object, access custom object property values from Python, and create basic primitives from defined details.
DM_Template3-PolyExtrude	Shows how to add toolbar items to DesignModeler, add custom objects to the tree, access custom object property values from Python, create a polygon using provided details, use the created polygon for extrude and resolve operations, and unite the create bodies to form a single-body object.
DM_Template4-ParametricPrimitive	Shows how to how to add toolbar items to DesignModeler, use the attribute `isparameter` to promote properties as DesignModeler design parameters, access property values from Python, and create a box primitive object.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

ACT Templates for DesignXplorer

The following table describes the folders in the template package for DesignXplorer. Each folder contains an extension example. For information about using ACT to customize DesignXplorer, see the *ACT Customization Guide for DesignXplorer*.

Folder	Description
CppOptimizer	Shows how to implement an extension from existing C/C++ code. The interface **`IOptimizationMethod`** is implemented in IronPython as a wrapper to the C++ symbols, using the **`ctypes`** foreign function library for Python.
CSharpOptimizer	Show how to set up an optimization extension when the interface **`IOptimizationMethod`** is implemented in C#. While the optimization algorithm is also implemented in C#, you could use only the implementation **`IOptimizationMethod`** as an adapter to an existing implementation in C# or another language. A Microsoft Visual Studio 2010 project is provided.
CSharpSampling	Shows how to set up a sampling extension when the interface**`ISamplingMethod`** is implemented in C#. While the optimization algorithm is also implemented in C#, you could use only the implementation **`ISamplingMethod`** as an adapter to an existing implementation in C# or another language. A Microsoft Visual Studio 2010 project is provided.
PythonOptimizer	Shows how to define optimization features in an extension using IronPython. This example runs a simple random exploration of the parametric space, generating the requested number of points and returning the best candidates identified. Fully implemented in IronPython (no build required), this example is your sandbox. It demonstrates many optimization extension features and is the best example to start with. **`PythonOptimizer`** is running a simple random exploration of the parametric space, generating the number of points requested by the user and returning the best candidates found.
PythonSampling	Shows how to define sampling features in an extension using IronPython. This example runs a simple random exploration of the parametric space, generating the requested number of points. Fully implemented in IronPython (no build required), this example is your sandbox. It demonstrates many sampling extension features and is definitely the best example to start with. **`PythonSampling`** is running a simple random exploration of the parametric space, generating the requested number of points.

ACT Templates for Mechanical

The following table describes the folders in the template package for Mechanical. Each folder contains an extension example. For information about using ACT to customize Mechanical, see the *ACT Customization Guide for Mechanical*.

Folder	Description
Template1-APDLMacroEncapsulation	Shows how to encapsulate an APDL macro within one ACT load object.
Template2-GenericObjectionCreation	Shows how to implement ACT generic objects in the model tree above analyses and how to implement an object that does not affect the solution state.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Folder	Description
Template3-TabularData	Shows how to implement an ACT property with tabular data.
Template4-CustomUnit	Shows how to implement an ACT load with a property having a non-standard unit.
Template5-ExternalSolve	Shows how to implement an ACT result that executes an APDL macro on an external Ansys solver process.
Template6-GenericPostObject	Shows how to implement ACT generic objects in the solution tree of the analyses.
Template7-PullDownMenus	Shows how to implement an ACT load with properties having pull-down menus.
Template8-TargetedAnalysisLoad	Shows how to implement an ACT load that is valid for the following simulation types: Static Structural, Modal, and Thermal.
Template9-Attributes	Shows how to use ACT attributes to save persistent data.
Template10-Graphics	Shows how to implement graphics features in an ACT extension.
Template11-Table_n_Graph	Shows how to implement a table and graph object using existing panes in Mechanical.
Template12-BodyView	Shows how to implement objects that show single-body and double-body views.
Template13-SolverBypass	Shows how to implement a solver that will allow native loads under it to have custom functions for property values where functions are applicable, and how to access those functions in the **<OnSolve>** callback.

ACT Wizard Templates

The following table describes the folders in the template package for ACT simulation wizards (p. 141). Each folder contains an extension example, with the exception of **Templates-FluentWizard**, which contains two Fluent extension examples.

Folder	Description
Template-DesignModelerWizard	Shows how to create a target product wizard for DesignModeler. This wizard creates a geometry.
Template-ElectronicsDesktopWizard	Shows how to create a target product wizard for Electronics Desktop. This wizard displays a list of available projects or designs and creates a cube.
Template-MechanicalWizard	Shows how to create a target product wizard for Mechanical. This wizard create the mesh for the geometry, adds boundary conditions, and then solves and displays a result.
Template-MixedWizard	Shows how to create a mixed wizard for multiple contexts. This wizard creates a static structural system, opens either DesignModeler or SpaceClaim, and creates a geometry (a cube). It then opens Mechanical, creates the mesh, defines

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Folder	Description
	the boundary conditions, and displays the maximal deformation.
Template-ProjectSchematicWizard	Shows how to create a project wizard for the Workbench **Project** tab. This wizard shows all the property capabilities for ACT. It also show how to include reports and charts.
Templates-FluentWizard	Shows how to create target product wizards for importing MSH and CAS files into Fluent. The extension **`FluentDemo1`** shows how to import a MSH file. The extension **`FluentDemo2`** shows how to create a CAS file. After the import, both extensions show how to run a simple analysis, generates results, and display the results in a customized panel in Fluent. For more information, see Fluent Wizard (MSH Input File) and Fluent Wizard (CAS Input File).
Template-SpaceClaimWizard	Shows how to create a target product wizard for SpaceClaim. This wizard shows all the property capabilities for ACT. It also show how to include reports and charts.

ACT Workflow Templates

The following table describes the folders in the package for ACT workflow templates. Each folder contains an extension example. For information about using ACT to customize Workbench, see the *ACT Customization Guide for Workbench*.

Folder	Description
CaptionChanger	Shows the dynamic update of task group captions.
CommandCancellation	Shows how to cancel a command from script.
Connections	Shows the filtering of connections with downstream target tasks. The *general transfer* type supports connectivity with **Model**, **Setup**, and **Solution** cells. Here, **Setup** and **Solution** connections are blocked, allowing only the **Model** connection to be established.
CSharp	Shows a simple workflow-based app that is coded using C#.
CustomCFD	Shows two custom CFD-based workflow task groups that mimic the integration of external processes. The final connection links the workflow with CFX.
CustomContextMenu	Shows the creation, registration, and filtering of a custom context menu. The menu appears when the right mouse button is clicked on a **Model** task of any Ansys-based analysis system.
CustomContextMenu2	Shows the creation, registration, and filtering of a custom context menu on a custom task.
CustomGeometry	Shows a custom CAD generation process integrated into Workbench. The geometry can then be passed into a downstream system for simulation setup and analysis.
CustomLoad	Shows the dual-work of a workflow extension (to interact with Workbench) and Mechanical extension (to operate on your behalf inside

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Folder	Description
	Mechanical). This example could form the basis of custom external load processing.
CustomPropertyAction	Shows the customization of an existing property. It changes the property into a button-based, dialog box retrieval of the property value.
CustomStructural	Shows a custom workflow-based analysis system (task group) that re-uses existing Mechanical-based tasks.
CustomTab	Shows the creation of a custom project tab in Workbench. You can set the tab control to a control provided by the UI Toolkit. For example, you can set the control to a web page viewer. The supplied IronPython script shows a custom view with HTML content.
CustomTab2	Shows the creation of a custom project tab in Workbench. You can set the tab control to a control provided by the UI Toolkit. For example, you can set the control to a web page viewer. The supplied IronPython script shows the re-use of built-in and custom views in a custom pane and displays only the parameters associated with the edited task. Each task has its own custom tab. Switching between tabs shows unique data loaded for each task tab.
CustomTransfer	Shows the transfer of data between two custom tasks using custom input and output data types.
DataSquares	Shows a simple data-integrated external processor as a custom task group having one task.
DemonstrationSovler	Shows a custom solver integrated using ACT. It also shows how to use an upstream custom task to provide custom data over ACT's general transfer mechanism. A toolbar button inside Mechanical displays the custom data retrieved from the upstream task.
Dictionary	Shows dictionary use for a task property.
DynamicContextMenus	Shows the creation, registration, and filtering of dynamic custom context menus on a custom task.
DynamicParameters	Shows how to dynamically create parameters from a context menu exposed from a task.
EmbeddedEXE	Shows how to embed an external application inside a Workbench pane.
Events	Shows framework event processing inside an empty extension.
GenericMeshTransfer	Shows a custom workflow that passes an upstream mesh to a downstream consumer. This could be a mesh-morphing integration.
GenericTransfer	Shows the use of ACT's generic transfer capability between two custom components. It uses the **`TransferData`** data store from which to push and pull data.
ImportedLoad	Shows an imported load being read from external files into an ACT task and fed into a Mechanical-based analysis **Setup** task. The behavior leverages external data-like behavior and relies on one of the already-supported external load types for boundary conditions. For custom types, see the **`CustomLoad`** template.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Folder	Description
InputGroups	Shows consuming from more than one type of source. The custom task can consume from a Mechanical-based **Solution** cell and a Fluent-based **Solution** cell.
LicenseCheck	Shows Ansys license API invocation from a toolbar button.
ObjectTest	Shows how you can use custom classes as property types on tasks. ACT relies on serialization support to persist property values. At the same time, Workbench relies on spec-based metadata to properly save its projects. Combining the two requires a little coding finesse and the need to use C# to describe the custom class definition. This is an advanced example to be used only in projects that must use custom property types.
Parameters	Shows the Workbench integration of a custom optimizer via a parametric task group.
PasswordProperty	Shows the customization of an existing property. You change the property into a button-based, dialog setting/retrieval of the property value to display a masked password task **`propertyS=`**.
Progress	Shows progress monitor API interaction to display messages during updates to Workbench's **Progress** pane.
ProjectPruner	Shows a toolbar button that allows users to select "keep" systems and delete all remaining in the project.
Properties	Shows task-based property manipulation for selection-based properties.
PythonModules	Shows a Python module-only load to access custom modules from the console or script.
Quantity	Show all possible declarations for task-defined quantity-based properties.
RSTProcessor	Shows a custom task that consumes from an upstream Ansys analysis solution task. Possible uses include custom postprocessing of the results from the generated RST file.
SaveAsHandler	Shows how to detect "save-as" events inside Workbench.
SDKWrapper	Shows the wrapping of an SDK integration with an ACT extension for https://catalog.ansys.com distribution.
Squares	Shows a simple external-connection-like integration of a parameters-only process-oriented task group.
TaskColonizer	Shows custom task background color selection via a custom context menu. The menu appears when your right-click any task in the Workbench **Project Schematic**. Color data is stored within the extension.
TaskGroup	Shows the most basic workflow implementation. The single-task task group has one callback (update), no properties, and no connections.
ToolBar	Shows custom Workbench toolbar buttons. Workbench does not support toolbar captioning. A no-operation, empty entry is used to display a caption.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Ansys Store Extensions

The Ansys Store provides hundreds of apps that further expand the capability of Ansys solutions. An Ansys app is simply a published ACT extension for performing specific functions within a targeted Ansys product.

The Ansys Store provides both free and paid apps developed by Ansys and trusted partners. While you are required to request a quote for a paid app for offline fulfilment, you only need to log in to download a free app. The Ansys Store supports filtering apps based on target application, product version, and price.

Download free apps for your licensed Ansys products to witness firsthand how sophisticated ACT extensions can be. If an app description concludes with **[Contains source code]**, the downloaded app includes a directory containing the extension's XML definition file and IronPython scripts. You can view and even copy code from these files into the files for your own custom ACT extensions.

For your convenience, the following table lists free apps that provide source code. The Ansys Store indicates the Ansys product versions in which an app is supported.

Ansys App	Targeted Product	App Description
API 4F Wind Loading	Mechanical	Applies wind loading on a solid, shell, or beam geometry per the API 4F guideline.
Advanced Enclosure	DesignModeler	Creates a fluid enclosure and decomposes it to sweepable bodies.
Analysis Settings Importer	Mechanical	Adds support for using a CSV file to import analysis settings for setting step end times, toggle automatic time stepping, and define substeps.
Command Obj to ACT	Mechanical	Converts your APDL command objects and snippets into working ACT extensions. This app makes it easy to create custom apps with your own specific APDL code.
Convert to Point Mass	Mechanical	Enables one-click conversion of solid components into point mass with automatic calculations and assignments of total mass, center of gravity location, and moment of inertia.
Explicit Time Step	Mechanical	Estimates and plots the Courant-Friedrichs-Lewy (CFL) time step for explicit dynamics analyses, based on the currently defined mesh and material properties, before executing the solution.
Extended Probe	Mechanical	Inspects deformed positions, angles, and distances between nodes.
External CAD Bridge	Workbench Workflow	Demonstrates how to create an app that integrates parametric study for non-Ansys CAD modeling programs.
Follower Loads	Mechanical	Creates follower forces and moments to follow geometric deformation.
General Axisymmetric Converter	Mechanical	Converts 2D geometry to 3D general axisymmetric elements (geometric axisymmetry with non-axisymmetric loading). For general features, see APDL element 272 or

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Ansys App	Targeted Product	App Description
		273. This app also introduces custom constraint for rigid constraint equations for working with new axisymmetric sections.
Honeycomb Creator	DesignModeler	Creates honeycomb core models from a few simple cell properties and bulk size specifications.
Hot Keys Customizer	Mechanical	Allows you to assign your own scripted actions to hot keys to make performing repetitive actions easy. This app can also call functions from other apps. This is demonstrated by calling functions from the app **Mechanical Custom Interface**, also described in this table.
MATLAB Optimizers for DX	DesignXplorer	Exposes MATLAB optimization algorithms and user programs in the **Optimization** cell of a DesignXplorer system.
Mechanical Custom Interface	Mechanical	Works in conjunction with the app **Hot Keys Customizer** to introduce custom and more efficient workflows for Mechanical.
Modal Mass	Mechanical	Allows you in a modal analysis to easily view the effective mass data without searching through the solver output file.
Motion Loads	Mechanical	Applies motion loads issued from a rigid body analysis in a static structural analysis.
Moving Heat Source	Mechanical	Facilitates the definition of a moving heat source in Mechanical.
Multi Object Export	Mechanical	Exports results for multiple objects at once, with the data combined into a single output file. This avoids the need for multiple exports and manual collection or combination of the data.
Porous Zone Calculator	Fluent	Calculates flow through a variety of porous models. You can easily compute permeability and inertial resistance coefficients by providing velocity and pressure drop profiles from a CSV file.
Rotordynamics Tools	Mechanical	Exposes extended functionality for importing bearing coefficients, plotting whirl orbits in graphics windows, getting Campbell diagram data, and setting up typical rotordynamic constraints.
Selection Toolbar	Mechanical	Provide several facilities to create or manipulate selections in Mechanical.
User Programmable Features	Mechanical	Allows for the use of User Programmable Features (UPF) within Workbench.
Windkessel	Fluent	Couples two- and three-element Windkessel models to hemodynamics simulations, allowing for realistic and customizable boundary conditions using Fluent.
nCode Fatigue Result	Mechanical	Displays results from nCode in Mechanical.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

ACT App Builder

The **ACT App Builder** is for creating ACT extensions in a visual environment, saving you from having to create or modify XML files and IronPython scripts directly. The following topics provide the information that you need for using the **ACT App Builder** for extension creation:

Starting the ACT App Builder

Opening and Creating App Builder Projects

Configuring an App Builder Project

Editing App Builder Entities

Testing Temporary Extensions

Importing and Exporting ACT Extensions

Using the XML Editor

Using the Resource Manager

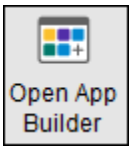
Note:

The **ACT App Builder** is not available on Linux platforms.

Starting the ACT App Builder

You can start the **ACT App Builder** from Workbench using either of the following methods:

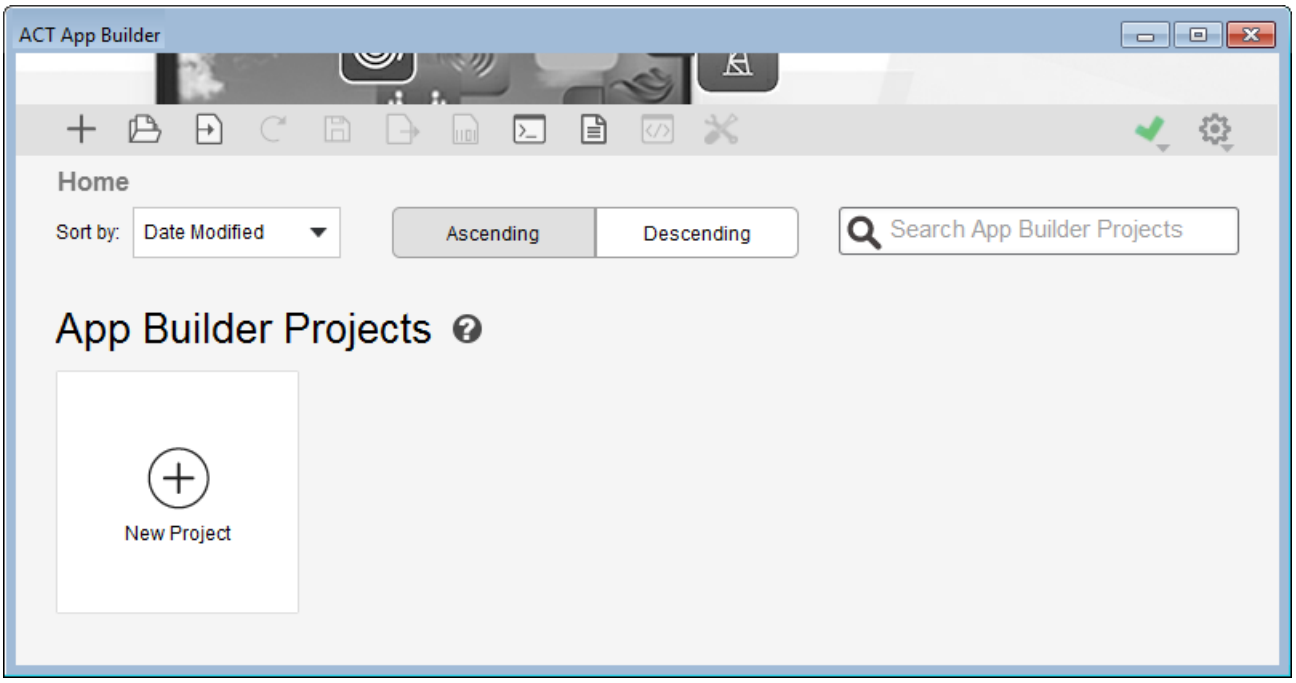
- Select **Extensions** → **Open App Builder**.
- From the ACT Start Page (p. 39), click **Open App Builder** in the toolbar.



Home Page

When the **ACT App Builder** starts, the **Home** page displays all app builder projects that you have created. An app builder project is a compressed file with an ACTPJ extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.



In the upper left corner of all **ACT App Builder** pages, the toolbar (p. 95) displays buttons for performing various actions. Under the toolbar, the navigation menu displays **Home** to indicate that you are currently on the initial page, where you can open (p. 98) or create (p. 98) an app builder project or import (p. 104) an ACT extension into an app builder project.

As your projects grow in number, you can use the sort and search capabilities (p. 97) near the top of the page to quickly find a particular project. On any page, you can click the help icon to display an overview and relevant workflow information.

ACT App Builder Toolbar

Descriptions follow for the buttons in the **ACT App Builder** toolbar. The buttons that are enabled depend on where you are in the app creation process.

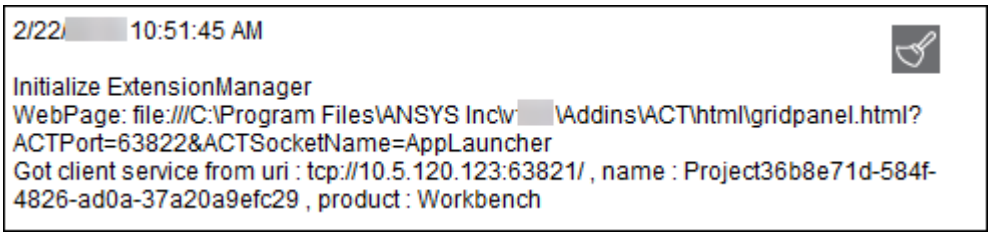
Button	Action
	Create (p. 98) a new app builder project.
	Open (p. 98) an existing app builder project.
	Import (p. 104) an ACT extension into a new app builder project.
	Upload (p. 103) the temporary extension for the active app builder project into the Ansys product for which it is being created so that you can test what has been created so far.
	Save the app builder project to a specified folder to which you can write. When you click this button, the changes that you have made in the **ACT App Builder** are saved to the ACTPJ file. If you take an action that might result in losing changes, the **ACT App Builder** warns that you did not save your changes. You can then choose whether to save or discard changes.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Button	Action
	Export (p. 107) the app builder project to a scripted ACT extension, thereby creating the XML extension definition file and IronPython script that makes up the extension.
	Open the binary extension builder (p. 49) so that you can save the app builder project as an extension in the binary format (WBEX file).
	Open the ACT Console (p. 53) in the active Ansys product so that you can test and debug commands in your IronPython scripts.
	Open the Extensions Log File (p. 47) so that you can see all information that has been written to this file.
 or 	Toggles between showing the XML editor (p. 108) and **ACT App Builder**. In the XML editor, you can view and modify all XML elements rather than just the basic elements displayed in the **ACT App Builder**.
 or 	Toggles between showing the resource manager (p. 111) and **ACT App Builder**. In the resource manager, you can view and manage the files in your ACT extension.

Two additional buttons appear to the right of the **ACT App Builder** toolbar.

- Clicking the first button displays the app builder log. The green check mark is shown if no errors or warnings exist. A red circle or yellow triangle with an exclamation point is shown if an error or warning occurs. You can click this button at any time to see all information that has been written to the app builder log. In the upper right corner of the log, the sweep button provides for clearing the existing log content.

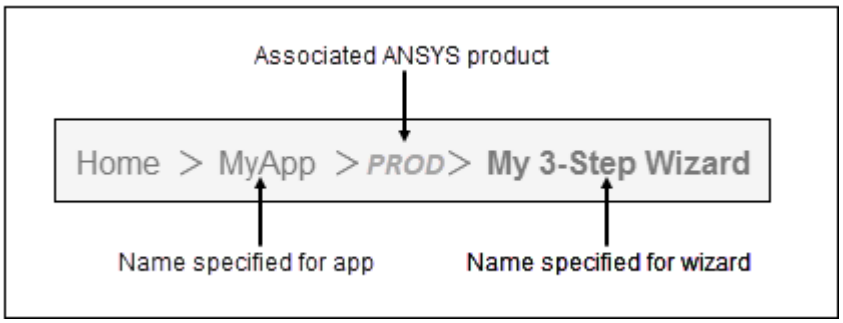


- Clicking the second button displays settings. The **Advanced Capabilities** check box is for turning on and off more complex entities of the **ACT App Builder** that are not yet visible in this version. The **Advanced XML Editor** check box is for turning on and off the display of additional non-published attributes in the XML editor. The **Language** option is for selecting the language for the user interface. After making a change, a notification displays in the newly selected language, indicating that you must restart the **ACT App Builder** for the language change to take full effect.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

ACT App Builder Navigation Menu

The **ACT App Builder** provides a navigation menu that consists of what is known as a *breadcrumbs trail*. The navigation menu keeps track of where you are in the creation process and provides links back to the previous pages through which you have navigated. For example, after creating an app for an Ansys product that has one wizard with three steps, the navigation menu would look similar to this:



To return to a previous page in the process, you simply click the link for the page. For example, after adding a wizard, you might need to go back to the page from which you added it so that you can you can either edit it or add another wizard. For more information, see Editing App Builder Entities (p. 103). To move to the next page in the process, you either click the block itself or the link for the next page in the navigation menu.

App Builder Sort and Search Options

As your app builder projects grow in number, you can sort them by the date modified (default), date created, or name. Clicking **Ascending** or **Descending** reverses the sort order.

You can also quickly locate a particular app builder project by typing text into the **Search App Builder Projects** option. Sort and search options display on the **Home** page and on the second page, where you add Ansys products to the app builder project.

Opening and Creating App Builder Projects

An app builder project is a compressed file with an ACTPJ extension. This compressed file contains folders with the files and services that are used to export the app builder project to an ACT extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Opening an App Builder Project

To open an existing app builder project, you do one of the following:

- On the **Home** page, locate the app builder project and click its block.
- In the toolbar, click the button for opening an app builder project  and then use the file browser to navigate to and select the existing project.

When the app builder project opens, you can use the navigation menu (p. 97) to move to the various pages in the creation process.

Creating an App Builder Project

To create an app builder project, you open and complete the **New Project** window.

1. To open this window, do one of the following:
   - On the **Home** page, click the **New Project** block.
   - In the toolbar, click the button for creating a new app builder project  .
2. In the **New Project** window, supply the name and destination folder for the app builder project.

   The name of the app builder project cannot contain spaces or certain characters and must be unique. The destination folder can be any folder to which you can write.
3. Click **OK**.

In the navigation menu (p. 97), the name of your newly created app builder project displays as the name of the second page. You can now configure the app builder project.

Configuring an App Builder Project

Configuring an app builder project consists of two primary steps:

Adding Ansys Products

Adding Wizards

Adding Ansys Products

On the second page of the **ACT App Builder**, you add the Ansys product for which you want to add a product feature. Currently, the **ACT App Builder** supports wizard creation for these products: DesignModeler, Electronics Desktop, Fluent, Mechanical, SpaceClaim, and Workbench.

To add the product, do the following:

1. Click the **Add Product** block. The **Add Product** window opens.
2. For **Product**, select the product for which you want to create one or more wizards.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

For example, select **Mechanical** if you want to create a wizard that is to run from Mechanical.

3. Click **OK**.

In the navigation menu (p. 97), the name of the selected product displays as the name of the third page. You can now add the ACT wizards that are to run from this product.

Adding Wizards

On the third page of the **ACT App Builder**, you add the wizards that are to run from the previously selected Ansys product. While the **ACT App Builder** currently supports only the wizard product feature, you can open the XML editor (p. 108) to view and add other features such as custom interfaces.

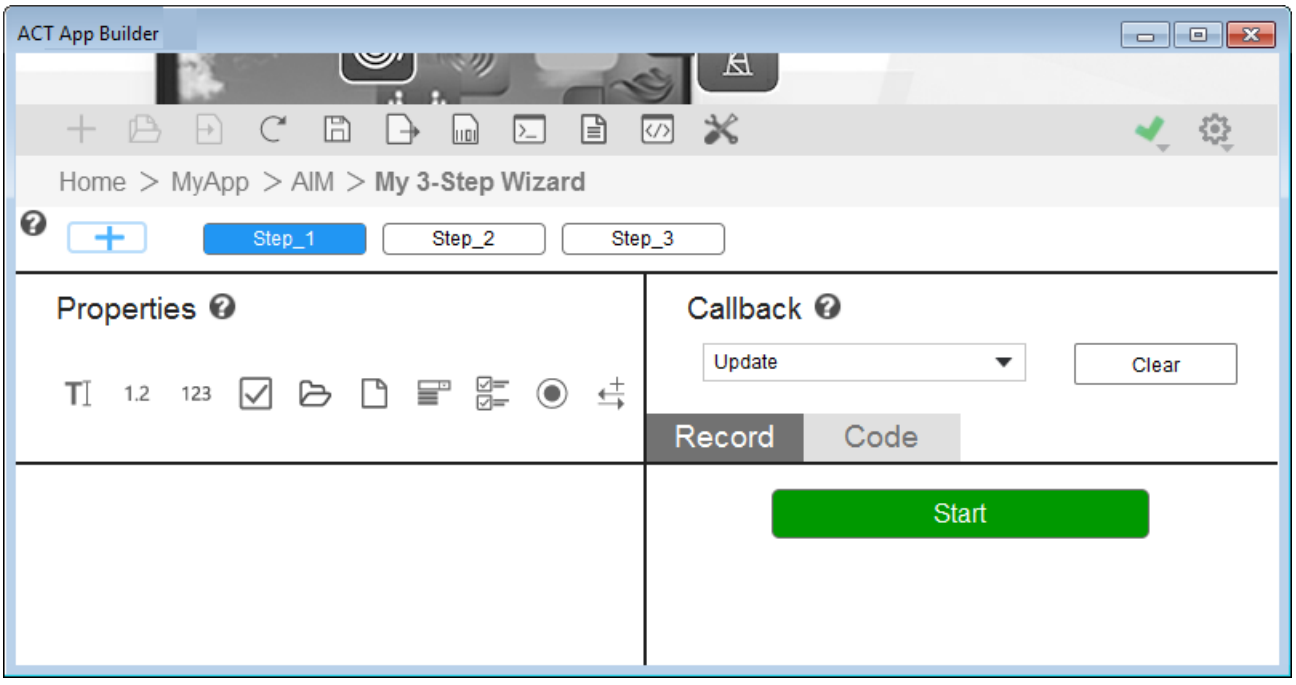
For each wizard to add, do the following:

1. Click the **New Wizard** block. The **New Wizard** window opens.
2. Supply the name, label, number of steps, and the icon to display.

   By default, the label is the name that you specify for the wizard.

3. Click **OK**.

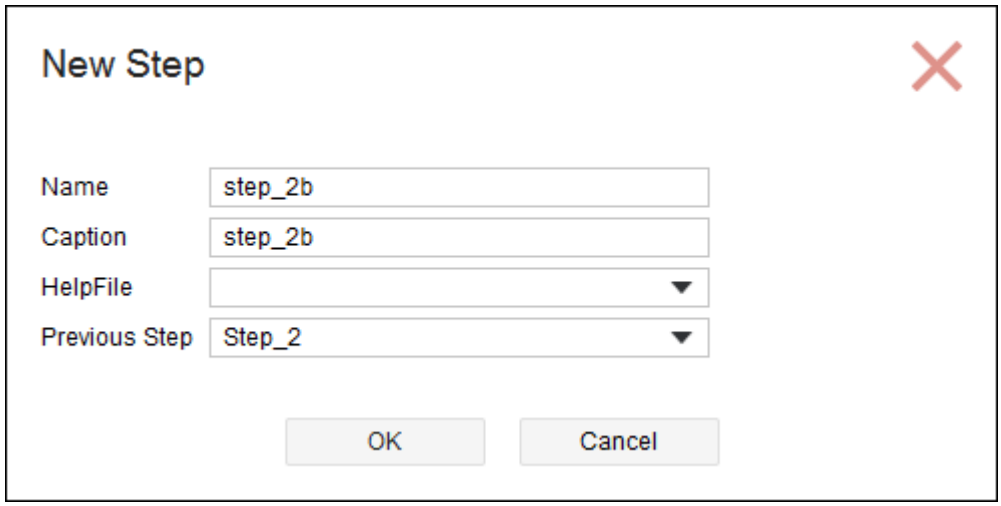
In the navigation menu (p. 97), the name of the wizard displays as the name of the fourth page. You can now define each of the steps for this new wizard by adding properties and callbacks.



This page has a step map for step creation and navigation in the upper left corner. It also has two panes for step definition: **Properties** and **Callback**. Clicking the help icons for each of these panes displays an overview and relevant workflow information.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- The step map provides buttons for navigating to existing steps and for inserting a new step. Clicking a button for an existing step displays its properties and callbacks. Clicking the plus sign button allows you to insert a new step after the existing step that you specify.

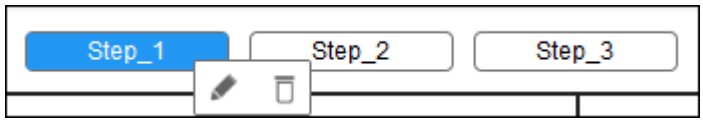


If you have first used the resource manager (p. 111) to import a folder with HTML files to use for custom help, for **HelpFile**, you can select the file that is to display for the step. All HTML files in the imported help folder are available for selection.

These files typically contain content that explain how to complete the steps. For more information, see Custom Wizard Help Files (p. 160).

In the XML editor (p. 108), you can also always use the optional attribute **`helpfile`** to assign HTML files to steps.

When you right-click a step in the step map, you can select a button for either editing or deleting the step.



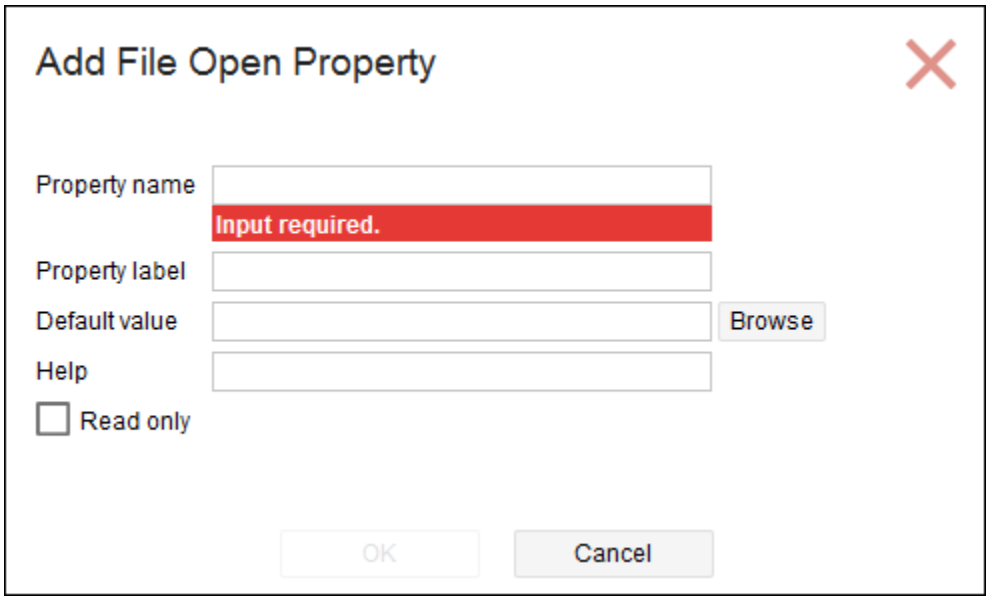
- The **Properties** pane displays a toolbar for defining the properties (p. 100) that the step uses.
- The **Callback** pane provides for defining the callbacks (p. 102) for the step. You typically use the **Record** tab to capture in a journal the actions that you take in the associated product. On the **Code** tab, you then view and edit the IronPython code recorded for callbacks to replace arguments with properties.

Defining Properties for a Wizard Step

Using the toolbar buttons in the **Properties** pane, you define the properties required for the active wizard step. For example, if the step is for opening a geometry file, you click the button for adding a file open property:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

You then define this property:



Tip:

For convenience, you can click a property in the toolbar and then drag and drop it into a particular location in the **Properties** pane. Additionally, you can drag and drop properties within this pane to rearrange them in the order in which you want them to display in the wizard.

In the main toolbar, clicking the button for uploading the temporary extension for the app builder project allows you to see the newly added properties when the wizard is loaded in the Ansys product. For more information, see Testing Temporary Extensions (p. 103).

Descriptions follow for all attributes that can possibly display for properties. The product with which a wizard is associated determines which toolbar buttons for properties display.

Attribute	Description
Group name	Name to assign to the property group. This property displays only for a group property.
Group label	Text to display for the property group in the wizard step. This property displays only for a slider property.
Property name	Name to assign to the property.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Attribute	Description
Property label	Text to display for the property in the wizard step.
Default value	Value to use if no value is entered or selected when this step of the wizard is run. For a property that opens a folder or file, click **Browse** to navigate to and select the desired folder or file.
Help	Help text to display for the property. This attribute provides for easily defining the help text for a property directly in the XML file. You can also choose to define property-level help in HTML files. For more information, see Custom Wizard Help Files (p. 160).
Options	List of values to display for a selection-based property. This property displays only for select, multiselect, and radio button properties.
Unit	Units for the property. This property displays only for float values.
Min	Minimum value to display for a slider control. This property displays only for a slider property.
Max	Maximum value to display for a slider control. This property displays only for a slider property.
Read-only	Indicates whether the property is read-only.

Defining Callbacks for a Wizard Step

A callback invokes an IronPython function. For each wizard step, you define the callbacks that can be invoked from this step in the **Callback** pane. For example, the callback **<onupdate>** specifies the function to invoke when **Next** is clicked to move to the next step. Similarly, the callback **<onreset>** specifies the function to invoke when **Previous** is clicked to return to the previous step to reset it if the step has not been completed correctly.

Note:

In the **Callback** pane, the list for selecting a callback includes choices such as **Update** and **Reset** rather than XML-formatted tags such as **<onupdate>** and **<onreset>**, which are used in the XML definition file that is created for the ACT extension.

You use the **Record** and **Code** tabs in the **Callback** pane to define callbacks.

- On the **Record** tab, you click **Start** to record in a journal the actions that you take to change project data in the interface of the associated Ansys product. Once you finish, you return to the **Record** tab and click **Stop**. To easily change the value of a journal entry, you can click it and then link the entry to a property that you have defined in the **Properties** pane.
- On the **Code** tab, you can view and modify the journal, which is recorded as an IronPython script. For example, you might need to replace arguments with properties. For information

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

on using journaling and scripting, as well as a complete list of all available data containers, namespace commands, and data types, see the *Workbench Scripting Guide*.

Note:

If you are recording and go to the **Code** tab, a message on this tab indicates that the code is currently read-only. If you want to make the code editable, you can click **Make Editable**. However, this will disable the recording.

When you click the toolbar button for saving the app builder project  , the IronPython code for the callback is saved.

Editing App Builder Entities

When you place the mouse cursor over a block for an app builder entity, a button with three vertical dots displays if the entity can be modified. Clicking this button displays a toolbar with edit and delete buttons. Similarly, toolbar buttons for editing and deleting display when you right-click a step, property, and so on.

Button	Action
	Edit the entity. When you click this button, the properties for the entity display so that you can make modifications.
	Delete the entity. When you click this button, the entity is deleted.

Tip:

To quickly change the property group for a property, you can drag and drop the property under a different group.

To easily edit properties that are not visible in the **ACT App Builder**, you can use the XML editor (p. 108).

Testing Temporary Extensions

As you use the **ACT App Builder**, you can see the changes live in Workbench by uploading the temporary extension for the app builder project. After clicking the toolbar button for uploading the

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

temporary extension , from the ACT Start Page (p. 39), click **Launch Wizards** to start the Wizards launcher (p. 47).

Importing and Exporting ACT Extensions

The **ACT App Builder** provides for importing an ACT extension as a new app builder project and exporting an existing app builder project as a new ACT extension. If you already have an ACT extension that is similar to the one that you want to create, you can import (p. 104) the existing extension into the **ACT App Builder** as a new app builder project, edit (p. 103) its entities, and then export (p. 107) the final app builder project to a new ACT extension, thereby creating its XML definition file and IronPython scripts.

Importing an ACT Extension

To import an ACT extension into an app builder project, do the following:

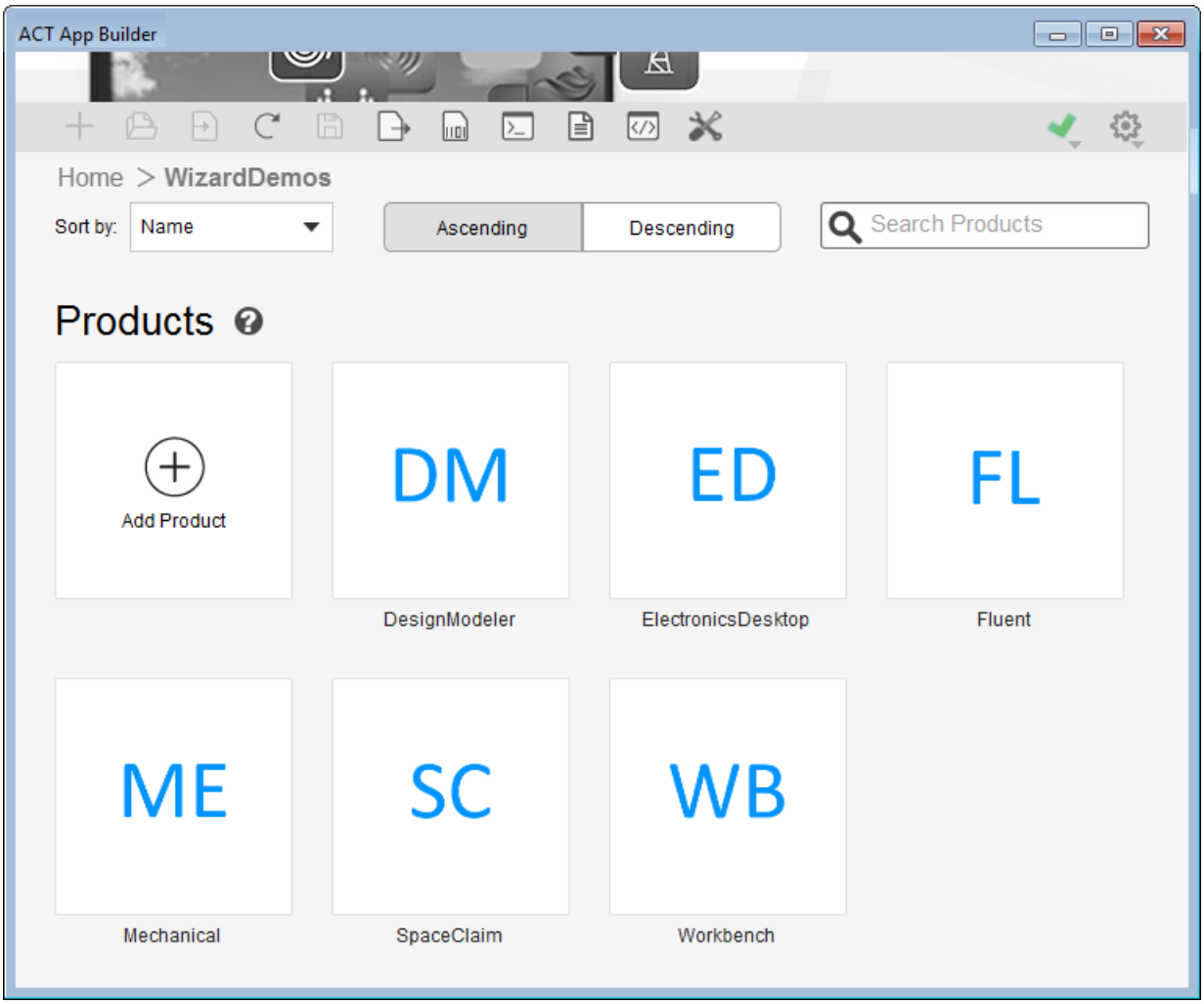
1. If you are not on the **Home** page, click **Home** in the navigation bar. If any unsaved changes exist in the currently open app builder project, you must indicate whether to save them before you can proceed.
2. In the toolbar, click the button for importing an extension . The **Import Extension** window opens.
3. Browse to and select the extension's XML definition file.
4. Supply the name and destination folder for the app builder project.

   The name of the app builder project cannot contain spaces or certain characters and must be unique. The destination folder can be any folder to which you can write.
5. Click **OK**.

The **ACT App Builder** imports the extension and opens it as an app builder project. A notification briefly appears at the top of the window when the import completes.

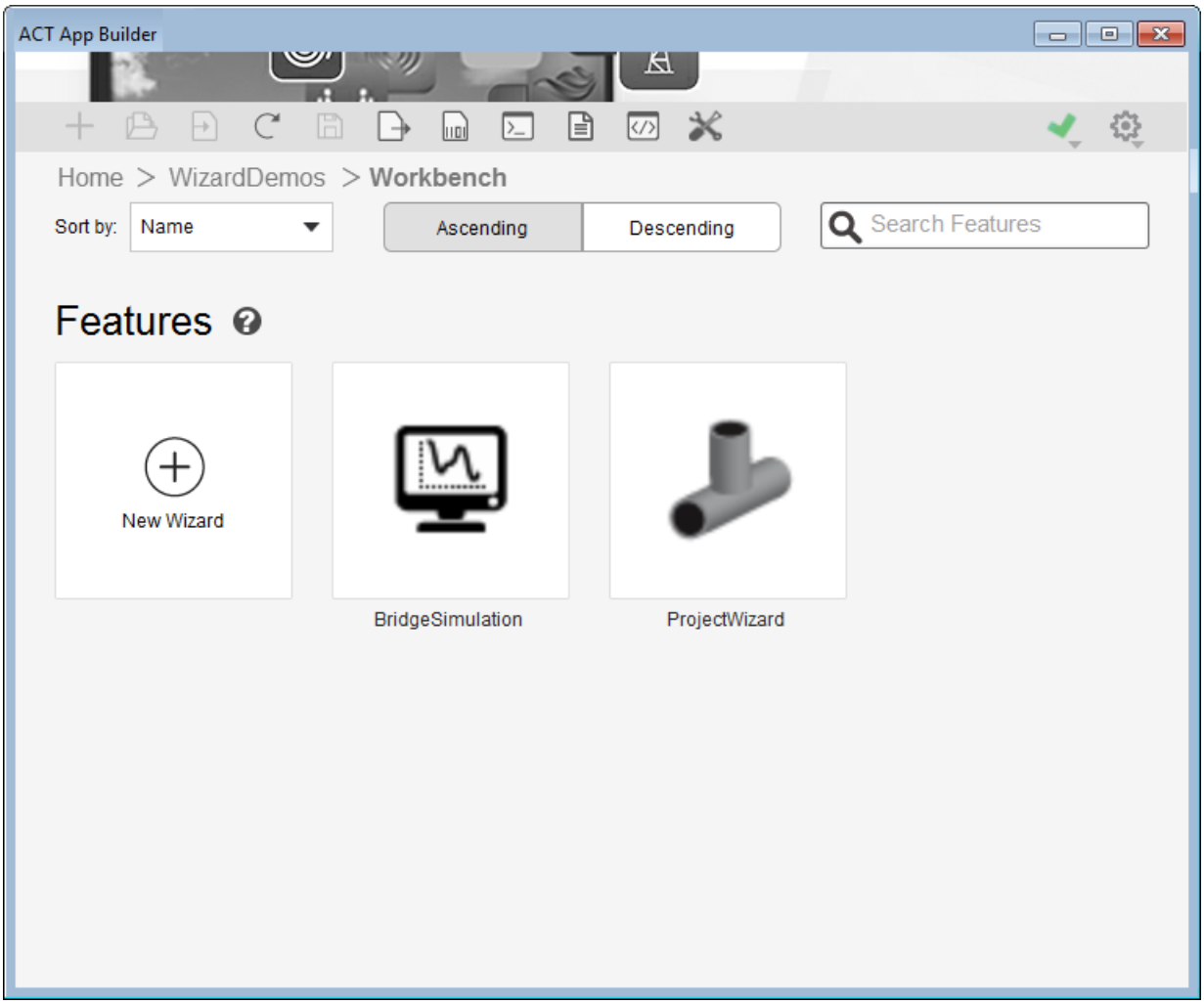
Because the supplied (p. 82) extension **`WizardDemos`** contains multiple target product wizards that are run from various Ansys products and a mixed wizard, it is often used for explanations. The following figure shows what the second page in the **ACT App Builder** looks like when this extension is imported. It has six products for which wizards have been added.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



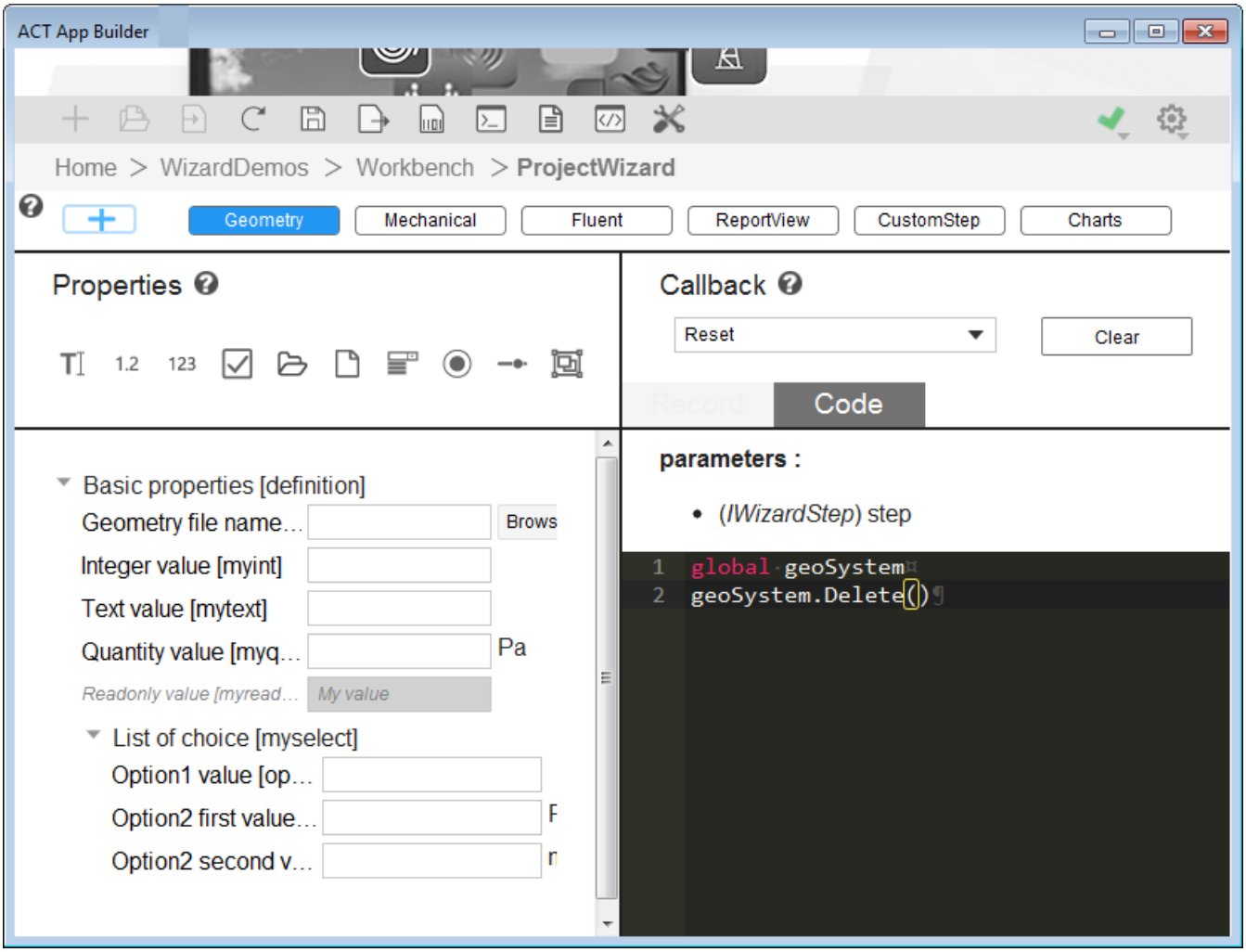
For example, clicking the **WB** block displays the two wizards associated with the Workbench product: **`BridgeSimulation`** and **`Project Wizard`**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Clicking the **ProjectWizard** block allows you to see the properties and callbacks defined for each of the six steps in this wizard, which is run from the Workbench **Project Schematic**. The following image displays properties for the step **`Geometry`** and the code for the callback **Reset**, which is the XML-formatted tag **`<onreset>`** in the XML file created for the extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



For more information about this particular extension, which has wizards that run from many different Ansys products, see Mixed Wizard Example (p. 155).

Exporting an ACT Extension

To export an app builder project to an ACT extension, do the following:

1. In the directory where you want to export the XML definition file and IronPython scripts, create a folder in which to export them.

2. In the **ACT App Builder** toolbar, click the button for exporting the extension . The **Select Folder** window opens.

3. After browsing to and selecting the folder created in the first step, click **Select Folder**.

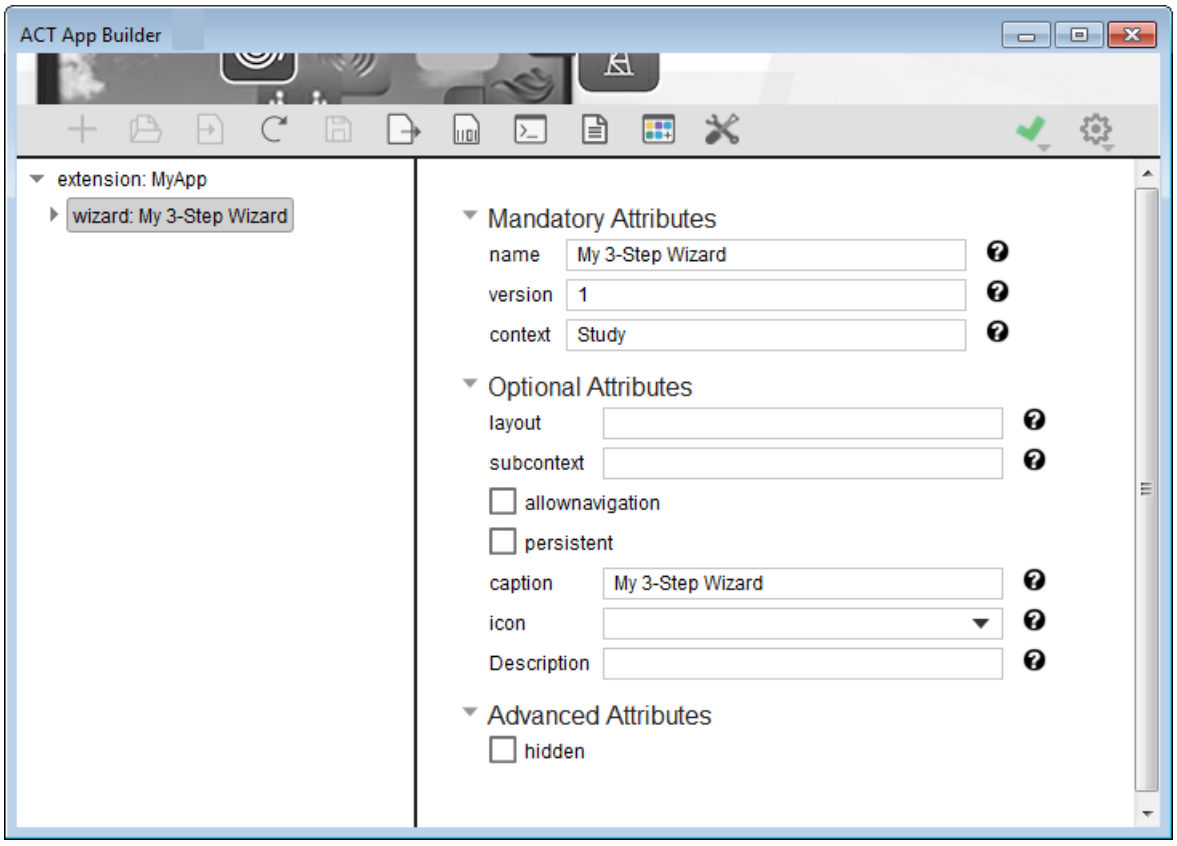
The XML definition file is created inside this folder. A child folder contains the IronPython scripts for the extension as well as additional folders with the extension's image files and HTML help files.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Using the XML Editor

To view or modify extension elements and attributes that do not display in the **ACT App Builder**, you click the second to last toolbar button to switch to the XML editor. While you are using the XML editor, this button is replaced with the button for switching back to the **ACT App Builder**.

The following figure demonstrates how the XML editor shows not only the basic wizard attributes that are visible in the **ACT App Builder** but also many more.

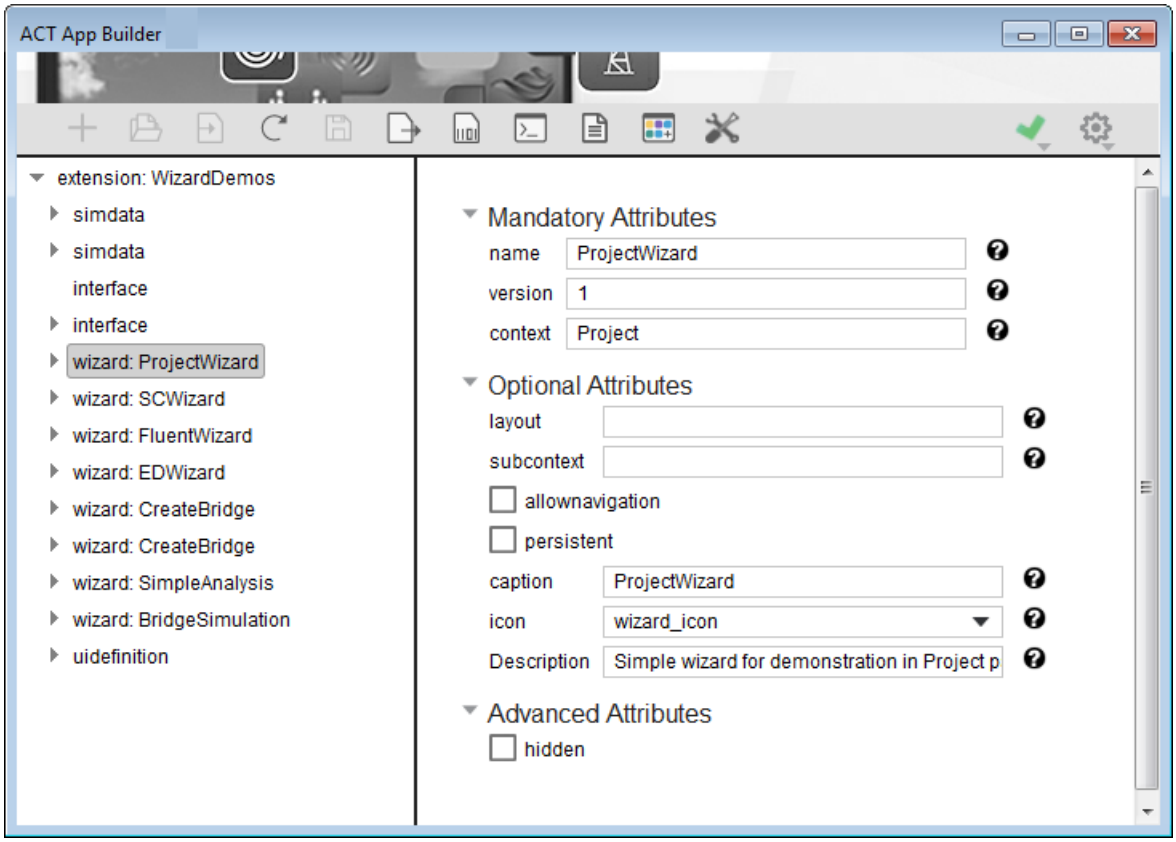


Tip:

You turn on and off the display of additional non-published attributes in the XML editor by clicking the settings icon in the upper right corner of the **ACT App Builder** and selecting and clearing the **Advanced XML Editor** check box. This figure displays the advanced attribute **`hidden`**.

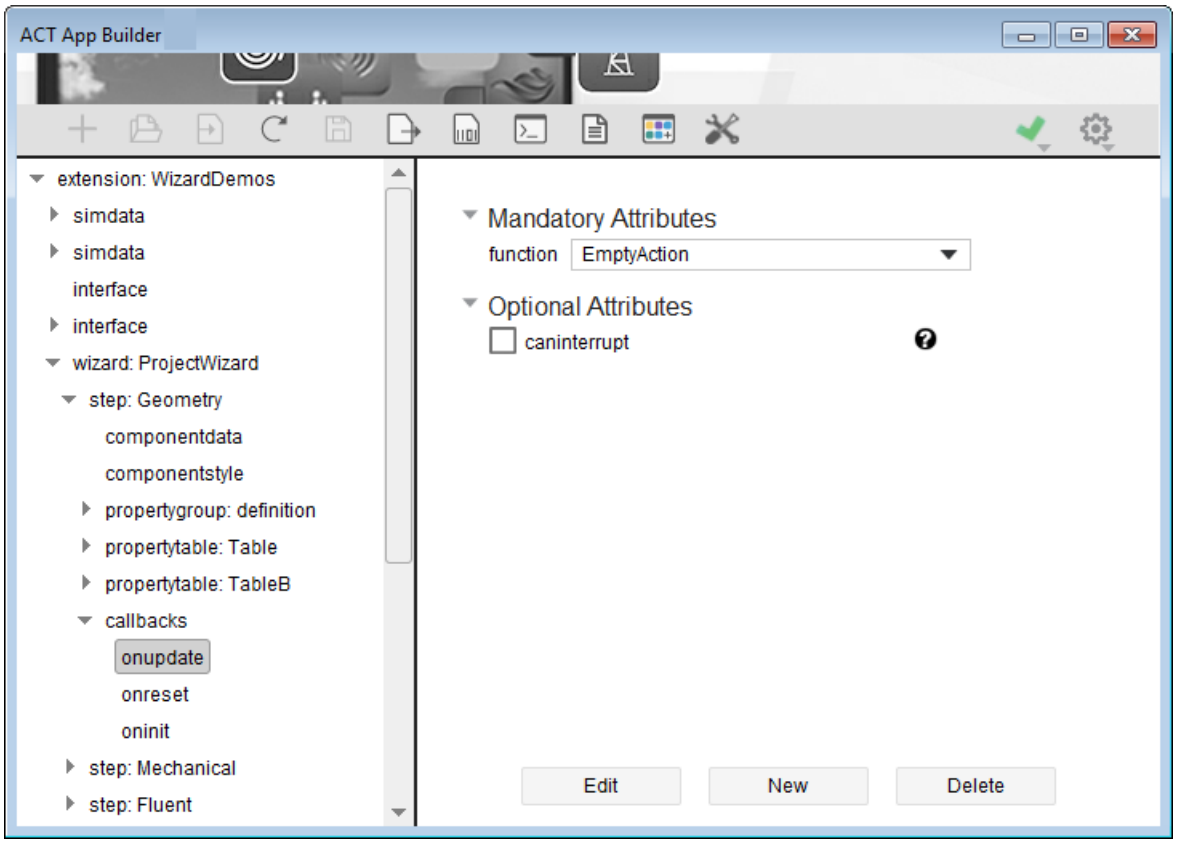
If you have imported the extension **`WizardDemos`** into the **ACT App Builder**, you can use the XML editor to view and modify all of its many elements, including **`<simdata>`**, **`<interface>`**, and **`<uidefintion>`**. The following figure shows the attributes for the Workbench wizard named **`ProjectWizard`**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

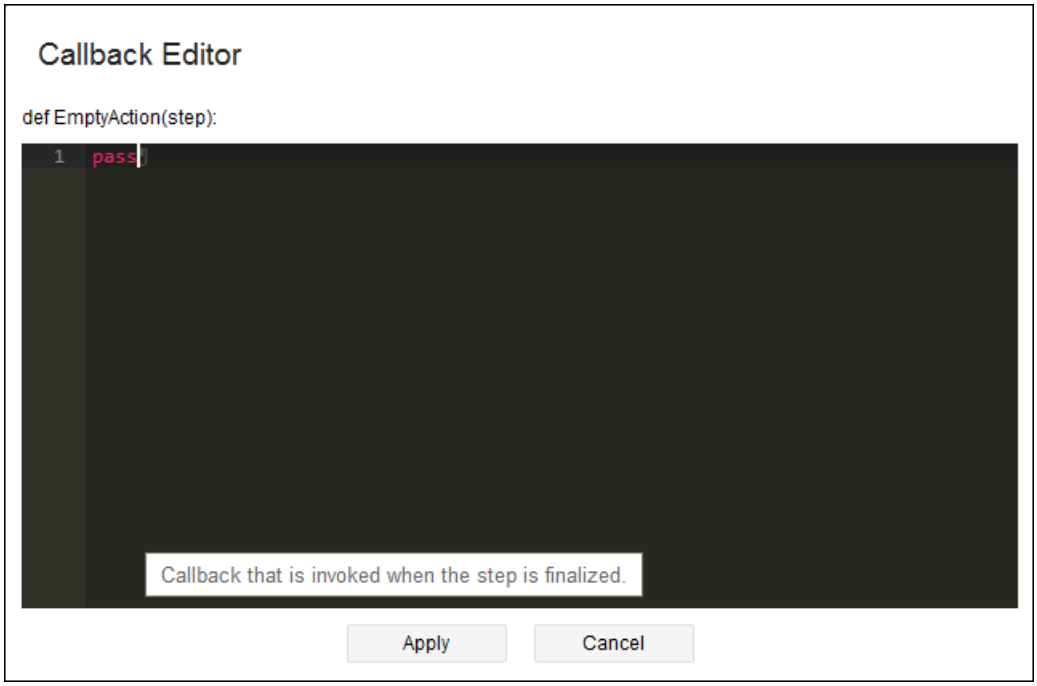


For example, if you expand the callback **`onupdate`** for the step **`Geometry`** in this wizard, you can view and modify the attributes for this step.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



When a callback for a wizard step is active in the XML editor, buttons in the lower portion of the window provide for editing, creating, and deleting attributes for the callback. Clicking **Delete** removes the selected attribute. Clicking **Edit** or **New** opens the **Callback Editor**, where you can view and modify the code.

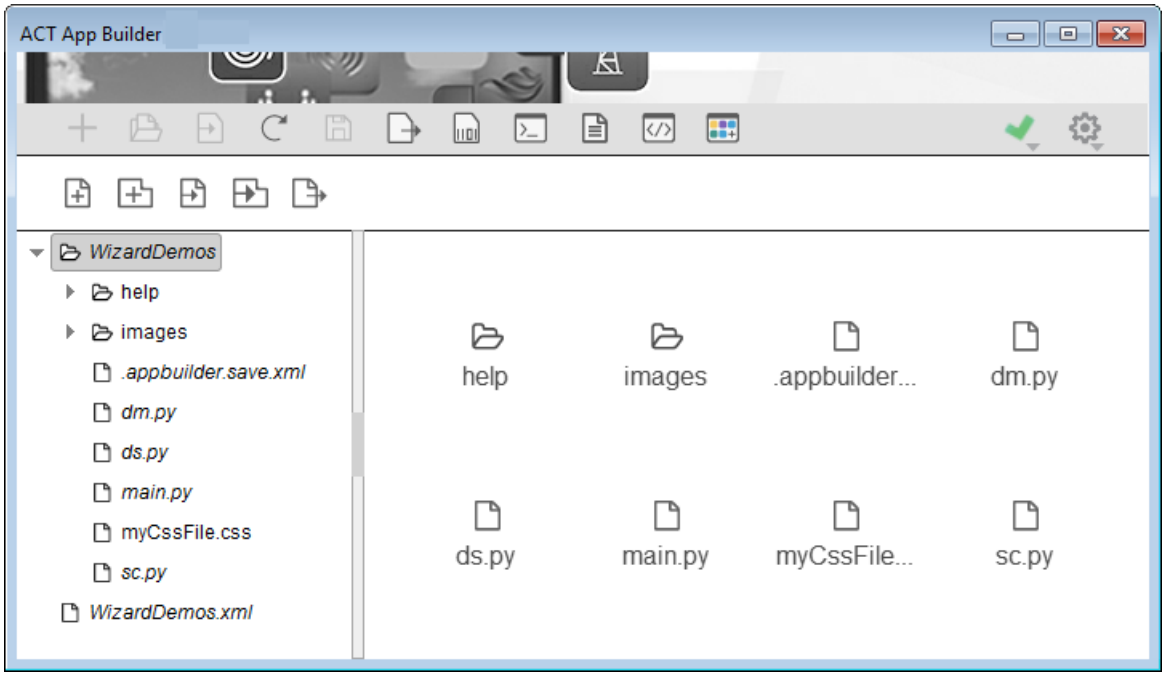


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

When finished, you click either **Apply** to save your changes or **Cancel** to discard them. You are then returned to the XML editor. When finished, you click the second to last toolbar button to switch back to the **ACT App Builder**.

Using the Resource Manager

To view and manage the files in your ACT extension, you click the last toolbar button to open the resource manager. While you are using the resource manager, this button is replaced with the button for switching back to the **ACT App Builder**.



The resource manager displays the folders and files managed by the **ACT App Builder** in italics to indicate that they are in use and are read-only. You cannot edit the name of the extension folder. You also cannot edit the app builder project's XML file or the extension's XML definition file or IronPython scripts.

Clicking a folder expands it so that you can see child folders and files. You can use drag-and-drop operations to organize folders and files. You can use the following toolbar buttons to add (p. 112), import (p. 113), and export (p. 114) files and folders.

Button	Action
	Add a new file. In the window that opens, specify the filename and folder location. You might use this option to add a new text file in which to enter information about the extension.
	Add a new folder. In the window that opens, specify the folder name and location.
	Import a file. In the window that opens, select the file to import.
	Import a folder. In the window that opens, select the folder to import. Importing a folder also imports all files within the folder.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Button	Action
	Export a file or folder. In the window that opens, select the file or folder to export. Exporting a folder also exports all files within the folder.

In the resource manager, you can edit the names of files and folders that do not display in italics. You can also edit the content of text files and custom HTML help files. For more information, see Editing Files and Folders (p. 113).

When finished, you click the last toolbar button  to switch back to the **ACT App Builder**.

Adding New Files and Folders

During extension creation, you can use the resource manager to easily add new files and folders. For example, you can add a text file in which to enter notes about your extension. You can also add a folder and then drag and drop existing files into this folder.

- To add a file, do the following:

  1. In the toolbar, click the button for adding a new file. The **Add New File** window opens.
  2. For **Name**, enter the name for the file, followed by the extension.

     For example, to create an HTML file to display as custom help for the first wizard step, you might enter **`step1.html`**.
  3. For **Location**, select the folder in which to add the file.

     For example, for an HTML file to display as custom help, you would select the child folder `help`. If this folder doesn't yet exist, before performing these steps, you would first want to add this folder by performing the steps indicated in the next bullet.
  4. Click **OK**.

- To add a folder, do the following:

  1. In the toolbar, click the button for adding a new folder. The **Add New Folder** window opens.
  2. For **Name**, enter the name for the folder.
  3. For **Location**, select the folder under which to add the new folder.
  4. Click **OK**.

You add content to a new file by opening it in an editor. For more information, see Editing Files and Folders (p. 113).

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Importing Existing Files and Folders

If files and folders for an extension exist elsewhere, you can use the resource manager to import them. For example, you can import a single image file or a folder with many image files from an existing extension into the current extension.

- To import a file, do the following:

  1. In the toolbar, click the button for importing a file. The **Import Existing File** window opens.
  2. For **Path**, click **Browse** to navigate to and select the file that you want to import.

     The name of the selected file appears for **Name**. You can rename the file if you want.
  3. For **Location**, select the folder in which to add the file.
  4. Click **OK**.

- To import a folder and all the files within it, do the following:

  1. In the toolbar, click the button for importing a folder. The **Import Existing Folder** window opens.
  2. For **Path**, click **Browse** to navigate to and select the folder that you want to import.

     The name of the selected folder appears for **Name**. You can rename the folder if you want.
  3. For **Location**, select the folder under which to add the imported folder.
  4. Click **OK**.

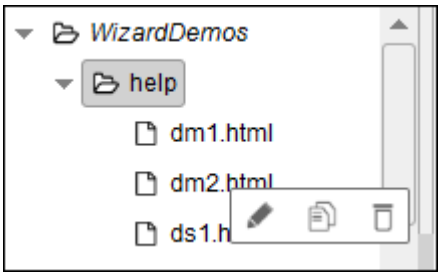
  The folder and its files display in the left pane as they are imported.

You can edit the names of imported files and folders and edit content in certain files by opening them in an editor. For more information, see Editing Files and Folders (p. 113).

Editing Files and Folders

From the resource manager, you can edit the names of folders and files that do not display in italics. The names that display in italics cannot be edited because the **ACT App Builder** is using them.

When you right-click a file or folder that can be edited, a toolbar displays.



Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

You use the following buttons to edit, duplicate, and delete the selected file or folder.

Button	Action
	Edit resource. When you click this button, the **Edit Resource** window opens so that you can edit the name of the file or folder.
	Duplicate resource. When you click this button, a copy of the resource is made. The new file or folder is given the same name as the original followed by a space and then the suffix **`(0)`**. For example, if you copied **`dm1.html`**, the newly created file would be named **`dm1 (0).html`**. If you copied **`dm1 (0).html`**, the newly created file would be named **`dm1 (0) (0).html`**.
	Delete resource. When you click this button, the **Delete Resource** window opens. You then confirm or cancel the deletion by clicking **Yes** or **No**.

Exporting a Selected File or Folder

From the resource manager, you can export the currently selected file or a folder. Exporting a folder exports all files in the folder.

To export a file or folder, do the following:

1. In the directory where you want to export the folder or file, optionally create a new destination folder.
2. Select the file or folder to export.
3. In the toolbar, click the button for exporting the selected file or folder. The **Export Selected File or Folder** window opens.
4. For **Type**, select **Folder** or **Archive**.
   - Select **Folder** (default) if you want to export the selected file or folder to the path indicated in the next step.
   - Select **Archive** if you want to export the selected file or folder to a compressed file (ZIP) to the path indicated in the next step.
5. For **Path**, click **Browse** to navigate to and select the destination folder.

   The name of the selected file or folder appears for **Name**. You can rename the file or folder if you want.
6. Click **OK**.

   The file or folder or file is exported to the destination directory. A notification briefly appears at the top of the window when the export completes.

Libraries and Advanced Programming

As you develop extensions to customize Ansys products, you can use installed libraries and advanced programming to replace IronPython code with C# assemblies:

Function Libraries

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Numerical Library

Advanced Programming in C#

Note:

While C# assemblies are specifically referenced, you can use any language that creates .NET assemblies.

Function Libraries

Libraries of IronPython functions are installed with ACT to help you to customize Ansys products. In `C:\Program Files\ANSYS Inc\v222\Addins\ACT\libraries`), libraries exist in the following folders:

- `AppLauncher`
- `DesignModeler`
- `ElectonicsDesktop`
- `Fluent`
- `Mechanical`
- `Project`
- `SpaceClaim`
- `Study`

You can use the IronPython functions within these libraries to develop extensions more efficiently. If desired, you can interactively test them in the ACT Console (p. 53).

Query to Material Properties

Description

This library allows you to access to all material properties defined in **Engineering Data**. The material is defined for each body and can be retrieved using the geometry API.

Location

- `libraries/Mechanical/materials.py`
- `libraries/Project/materials.py`
- `libraries/Study/materials.py`

Usage

```
import materials
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Functions

```
GetListMaterialProperties(mat)
```

This function returns a list of property names for the material **mat** given in argument.

```
GetMaterialPropertyByName(mat, name)
```

This function returns the material property **name** for the material **mat**.

```
InterpolateData(vx, vy, vin)
```

This function computes a linear interpolation **vout = f(vin)** with f defined by **vy = f(vx)**.

- **vx** represents a list of values for the input variable.
- **vy** represents a list of values for the property that depends on the input variable defined by **vx**.
- **vin** is the value on which the function has to evaluate the property.

Examples

Assume that you enter these commands:

```
import materials
```

```
mat = ExtAPI.DataModel.GeoData.Assemblies[0].Parts[0].Bodies[0].Material
```

```
materials.GetListMaterialProperties(mat)
```

Results like these are returned:

```
['Compressive Ultimate Strength', 'Compressive Yield Strength', 'Density', 'Tensile Yield Strength',
'Tensile Ultimate Strength', 'Coefficient of Thermal Expansion', 'Specific Heat', 'Thermal Conductivity',
'Alternating Stress', 'Strain-Life Parameters', 'Resistivity', 'Elasticity', 'Relative Permeability']
```

Assume that you enter these commands:

```
prop = materials.GetMaterialPropertyByName(mat,"Elasticity")
```

```
prop
```

Results like these are returned:

```
{"Poisson's Ratio": ['', 0.29999999999999999, 0.29999999999999999, 0.29999999999999999],
'Bulk Modulus': ['Pa', 166666666666.667, 175000000000.0, 183333333333.333],
'Temperature': ['C', 10.0, 100.0, 1000.0], "Young's Modulus": ['Pa', 200000000000.0,
210000000000.0, 220000000000.0],
'Shear Modulus': ['Pa', 76923076923.0769, 80769230769.2308, 84615384615.3846]}
```

Assume that you enter these commands:

```
val = materials.InterpolateData(prop["Temperature"][1:],prop["Young's Modulus"][1:],10.)
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
val
```

```
val = materials.InterpolateData(prop["Temperature"][1:],prop["Young's Modulus"][1:],20.)
```

```
val
```

Results like these are returned:

```
200000000000.0
```

```
201111111111.0
```

Units Conversion

Description

This library implements a set of functions to manipulate the unit-dependent quantities within an extension. This library is of particular interest each time quantities have to remain consistent with the current unit system activated in the Ansys product.

Location

- `libraries/DesignModeler/units.py`
- `libraries/ElectonicsDesktop/units.py`
- `libraries/Fluent/units.py`
- `libraries/Mechanical/units.py`
- `libraries/Project/units.py`
- `libraries/SpaceClaim/units.py`
- `libraries/Study/units.py`

Usage

```
import units
```

Function

```
ConvertUnit(value,fromUnit,toUnit[,quantityName])
```

This function converts the **`value`** from the unit **`fromUnit`** to the unit **`toUnit`**. A quantity name can be specified to avoid conflict during conversion. However, a quantity name is not mandatory.

Example

Assume that you enter these commands:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
import units
units.ConvertUnit(1,"m","mm")
```

A result like this is returned:

```
1000.0
```

Function

```
ConvertUserAngleUnitToDegrees(api, value)
```

This function converts the angle **`value`** to the unit degrees. If the current activated unit is already degrees, then the value remains unchanged.

Example

Assume that you enter these commands:

```
import units
units.ConvertUserAngleUnitToDegrees(api,3.14)
```

A result like this is returned if the angle unit is set to radians when the command is called:

```
180
```

A result like this is returned if the angle unit is set to degrees when the command is called:

```
3.14
```

Function

```
ConvertToSolverConsistentUnit(api, value, quantityName, analysis)
```

This function converts the **`value`** of the quantity **`quantityName`** from the currently activated unit in the Ansys product to the corresponding consistent unit used by the solver for the resolution of the analysis **`analysis`**.

Example

Assume that you enter these commands:

```
import units
units.ConvertToSolverConsistentUnit(api,1.,”pressure”,”Static Structural”)
```

A result like this is returned if the unit system is set to Metric(mm,dat,N,s,mV,mA) when the command is called:

```
10
```

Function

```
ConvertToUserUnit(api, value, fromUnit, quantityName)
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

This function converts the **value** of the quantity **quantityName** from the unit **fromUnit** to the currently activated unit system in the Ansys product.

Example

Assume that you enter these commands:

```
import units
units.ConvertToUserUnit(api,1.,”m”,”Length”)
```

A result like this is returned if the current activated unit is millimeter when the command is called:

```
1000
```

Function

```
ConvertUnitToSolverConsistentUnit(api, value, fromUnit, quantityName, analysis)
```

This function converts the **value** of the quantity **quantityName** from the unit **fromUnit** to the consistent unit that is used by the solver for the resolution of the analysis **analysis**.

Example

Assume that you enter these commands:

```
import units
units.ConvertUnitToSolverConsistentUnit(api,1.,”m”,”Length”,”Static Structural”)
```

A result like this is returned if the consistent unit is millimeter when the command is called:

```
1000
```

Function

```
GetMeshToUserConversionFactor(api)
```

This function returns the scale factor to be applied to convert a length from the unit associated with the mesh and the currently activated unit in the Ansys product.

Example

Assume that you enter these commands:

```
import units
units.GetMeshToUserConversionFactor(api)
```

A result like this is returned if the unit associated with the mesh is meter and if the current activated unit in the application is millimeter:

```
1000
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Quantity Types

Here is the exact text to use for the available quantity types:

- Angle
- Chemical Amount
- Current
- Length
- Luminance
- Mass
- Solid Angle
- Temperature
- Time
- Electric Charge
- Energy
- Force
- Power
- Pressure
- Voltage

MAPDL Helpers

Description

This library implements some helpers to write APDL command blocks or to execute Mechanical APDL programs.

Location

- `libraries/ElectronicsDesktop/ansys.py`
- `libraries/Fluent/ansys.py`
- `libraries/Mechanical/ansys.py`
- `libraries/Project/ansys.py`
- `libraries/Study/ansys.py`

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Usage

```
import ansys
```

Functions

```
createNodeComponent(refIds,groupName,mesh,stream,fromGeoIds=True)
```

This function writes the APDL command block (CMBLOCK) for the creation of a node component related to the geometry entities identified by **`refIds`** into the stream **`stream`**. "refIds" refer to geometric IDs if "fromGeoIds" is true and the node IDs are retrieved from geometric entities using the associativity between the geometry and mesh. If "fromGeoIds" is false, then "refIds" refer directly to the node IDs to be written in the component.

```
createElementComponent(refIds,groupName,mesh,stream,fromGeoIds=True)
```

This function writes the APDL command block (CMBLOCK) for the creation of an element component related to the geometry entities identified by "refIds" into the stream "stream". "refIds" refer to geometric IDs if "fromGeoIds" is true and the element IDs are retrieved from geometric entities using the associativity between the geometry and the mesh. If "fromGeoIds" is false, the "refIds" refer directly to the element IDs to be written in the component.

```
RunANSYS(api,args,[runDir[,exelocation]])
```

This function calls the Mechanical APDL program with the command line initialized by **`args`**. The folder from which to execute the program can be specified with **`runDir`**. The location of the executable is also managed with **`exelocation`**.

The parameter **`api`** must be **`ExtAPI`**.

Example

```
import ansys
ansys.createNodeComponent([refId],"myGroup",mesh,stream)
ansys.RunANSYS(ExtAPI,"")
```

Journaling Helper

Description

This library implements a helper that can be used to transfer data between Mechanical and the Workbench **Project** tab.

Location

- `libraries/Mechanical/ansys.py`
- `libraries/Project/ansys.py`
- `libraries/Study/ansys.py`

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Usage

```
import wbjn
```

Functions

```
ExecuteCommand(api,cmd,**args)
```

This function executes a journal command specified by **`cmd`** You can get a result object by using the function `returnValue(obj)` in your journal command. All arguments must implement the serialization interface provided by .Net. The object sent to the function `returnValue(obj)` must also implement the serialization interface. This interface is already implemented for many standard types of data (integer, double, list…). Note that the standard Python dictionary does not implement this interface by default. If a dictionary is required, use the class **`SerializableDictionary`** provided by ACT.

The parameter **`api`** must be **`ExtAPI`**.

```
ExecuteFile(api,file,**args)
```

The same as above but executes a journal file specified by **`file`**.

Examples

The following commands return **`5`** as the result:

```
import wbjn
wbjn.ExecuteCommand(ExtAPI,"returnValue(a+b)",a=2,b=3)
```

The following commands return **`My first system is: Static Structural !`**:

```
import wbjn
wbjn.ExecuteCommand(ExtAPI,"returnValue(a+GetAllSystems()[0].DisplayText+b)",
a="My first system is: ",b=" !")
```

The following commands return **`3`** as the result:

```
import wbjn
dict = SerializableDictionary[str,int]()
dict.Add("e1",1)
dict.Add("e2",2)
wbjn.ExecuteCommand(ExtAPI,'returnValue(d["e1"]+d["e2"])', d=dict)
```

Numerical Library

The numerical library that IronPython supports is `Math.net`. This library is available for use with ACT and is installed in the following directory:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

`C:\Program Files\ANSYS Inc\v222\Addins\ACT\bin\Win64\MathNet.Numerics.dll`

Note:

Because `Numpy` and `SciPyY` are numerical libraries supported by Python rather than IronPython, they cannot be used with ACT.

The following code sample provides three examples for using `Math.net`:

```
import clr
import System
import os
clr.AddReferenceToFileAndPath("C:\\Program Files\\ANSYS Inc\\v222\\Addins\\ACT\\bin\\Win64\\MathNet.Numerics.dll

# Example 1
import MathNet
from MathNet.Numerics.LinearAlgebra import *
V=Vector[System.Double].Build
M=Matrix[System.Double].Build

m = M.Random(3,4)
v = V.Random(3)

r=v*m
print r

# Example 2
import MathNet.Numerics.LinearAlgebra as la

from System import Array as sys_array
def array(*x): return sys_array[float](x) #float is equivalent to .Net double

A = la.Double.Matrix.Build.DenseOfRowArrays(
   array(3, 2,-1),
   array(2,-2,4),
   array(-1,.5,-1)
)

b = la.Double.Vector.Build.DenseOfArray(array(1, -2, 0))
x = A.Solve(b)

print x

# Example 3
A1 = la.Double.Matrix.Build.DenseOfRowArrays(
   array(3.0, 4.0, -1.0, 0.0),
   array(4.0, 5.0, 0.0, -1.0),
   array(5.0, 6.0, 0.0, 0.0),
   array(6.0, 7.0, 0.0, 0.0)
)
b1 = la.Double.Vector.Build.DenseOfArray(array(0, 0, 20, 0))
x1 = A1.Solve(b1)

print x1
```

Advanced Programming in C#

C# provides two major advantages over IronPython:

- Better performance
- Superior development environment that provides auto completion

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

It is assumed that you already know how to create assemblies in C#. The following topics explains how to replace IronPython code with C# assemblies:

Initialize the C# Project

C# Implementation for a Load

C# Implementation for a Result

Note:

While C# assemblies are specifically referenced, you can use any language that creates .NET assemblies.

Tip:

Included in the supplied workflow templates (p. 89) is a folder **`CSharp`**. This folder contains files for a simple workflow-based app that is coded using C#. While this example is for Mechanical, the concept is supported ACT-wide. For download information, see Extension and Template Examples (p. 81).

Initialize the C# Project

Once you have created a C# project and associated the type **`Class Library`** with this project, add references to the **`Ansys.ACT.Interfaces`** and **`Ansys.ACT.WB1`** assemblies of ACT. Related DLLs for these assemblies are located in the following directories, where *`Platform`* is either `Win64` or `Linux64`, depending on your operating system:

- `C:\Program Files\ANSYS Inc\v222\Addins\ACT\bin\`*`Platform`*`\Ansys.ACT.Interfaces.dll`.
- `C:\Program Files\ANSYS Inc\v222\aisol\bin\`*`Platform`*`\Ansys.ACT.WB1.dll`.

C# Implementation for a Load

The following XML file declares a load to be created in Mechanical that requires C# implementation:

```
<extension version="1" name="CSharp">

  <author>Ansys</author>
  <description>This extension demonstrates how to use CSharp to write extension.</description>

  <assembly src="CSharp.dll" namespace="CSharp" />

  <interface context="Mechanical">

    <images>images</images>

  </interface>

  <simdata context="Mechanical">

    <load name="CSharpLoad" caption="CSharp Load" version="1" icon="tload" unit="Temperature"
          color="#0000FF" class="CSharp.Load">
      <property name="Geometry" control="scoping">
        <attributes>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
          <selection_filter>face</selection_filter>
        </attributes>
      </property>
    </load>

  </simdata>

</extension>
```

In the definition of the load object, the only change is the use of the attribute **`class`**. This attribute must be set to the name of the class to be used for the integration of the load.

A description of the class **`CSharp.Load`** follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ansys.ACT.Interfaces.Common;
using Ansys.ACT.Interfaces.Mesh;
using Ansys.ACT.Interfaces.Mechanical;
using Ansys.ACT.Interfaces.UserObject;

namespace CSharp
{
    public class Load
    {
        private readonly IMechanicalExtAPI _api;
        private readonly IMechanicalUserLoad _load;
        public Load(IExtAPI api, IUserLoad load)
        {
            _api = (IMechanicalExtAPI) api;
            _load = (IMechanicalUserLoad) load;
        }

        public virtual IEnumerable<double> getnodalvaluesfordisplay(IUserLoad load, IEnumerable<int> nodeIds)
        {
            var res = new List<double>();
            IMeshData mesh = _load.Analysis.MeshData;
            foreach (int nodeId in nodeIds)
            {
                INode node = mesh.NodeById(nodeId);
                res.Add(Math.Sqrt(node.X * node.X + node.Y * node.Y + node.Z * node.Z));

            }
            return res;
        }

    }
}
```

To implement a callback in C#, create a new method in your class with the name of the callback in lower case.

In the example, you implement the callback **`<getnodalvaluesfordisplay>`** by adding the method **`getnodalvaluesfordisplay`** to the class.

C# Implementation for a Result

The following XML file declares a result to be created in Mechanical that requires C# implementation:

```
<extension version="1" name="CSharp">

  <author>Ansys</author>
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
  <description>This extension demonstrates how to use CSharp to write extension.</description>

  <assembly src="CSharp.dll" namespace="CSharp" />

  <interface context="Mechanical">

    <images>images</images>

  </interface>

  <simdata context="Mechanical">

    <result name="CSharpResult" caption="CSharp Result" version="1" unit="Temperature" icon="tload" location="
      <property name="Geometry" control="scoping" />
    </result>

  </simdata>

</extension>
```

For the load definition, the attribute class must be set to the name of the class to be used for the integration of the result.

A description of the class **`CSharp.Result`** follows.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Ansys.ACT.Interfaces.Common;
using Ansys.ACT.Interfaces.Mesh;
using Ansys.ACT.Interfaces.Mechanical;
using Ansys.ACT.Interfaces.UserObject;
using Ansys.ACT.Interfaces.Post;

namespace CSharp
{
    public class Result
    {
        internal double[] res = new double[1];

        public Result(IMechanicalExtAPI extApi, IUserResult result)
        {
        }

        public void evaluate(IUserResult entity, IStepInfo stepInfo, IResultCollector collector)
        {
            foreach (var id in collector.Ids)
            {
                 res[0] = id;
                 collector.SetValues(id, res);
            }
        }
    }
}
```

As for the load definition, the implementation of a new callback simply requires you add a new method with the name of the callback.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Feature Creation

Feature creation is the direct, API-driven customization of Ansys products. In addition to leveraging the functionality already available in a product, ACT enables you to add functionality and operations of your own. Examples of feature creation include creating custom loads and geometries, adding custom preprocessing or postprocessing features, and integrating third-party solvers, sampling methods, and optimization algorithms.

The following Ansys products support feature creation:

- DesignModeler
- DesignXplorer
- Mechanical
- Workbench

Capabilities common to all of these products include:

Toolbar Creation

Dialog Box Creation

Extension Data Storage

ACT-Based Property Creation

ACT-Based Property Parameterization

Note:

- Extensions referenced in this section are supplied (p. 81). In the extension's XML file, the element **`<interface>`** has an attribute **`<context>`**. This attribute specifies the Ansys product in which the extension is to execute. For most of the supplied extensions, the attribute **`<context>`** is set to **`Mechanical`**. The only significant product difference is that Mechanical requires BMP files for the images to display as toolbar buttons. All other products require PNG files.
- For feature creation capabilities specific to a product, see the ACT customization guide for the particular product. Links to these guides are available in ACT Customization Guides for Supported Ansys Products (p. 183).

Toolbar Creation

You can add custom toolbars to Ansys products that expose their own toolbars. A toolbar is a parent container for one or more toolbar buttons. Conceptually, the toolbar should address a specific feature,

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

with each toolbar button being associated with a function that supports the feature. This relationship is reflected in the structure of the extension's XML file, which defines the toolbar and its buttons.

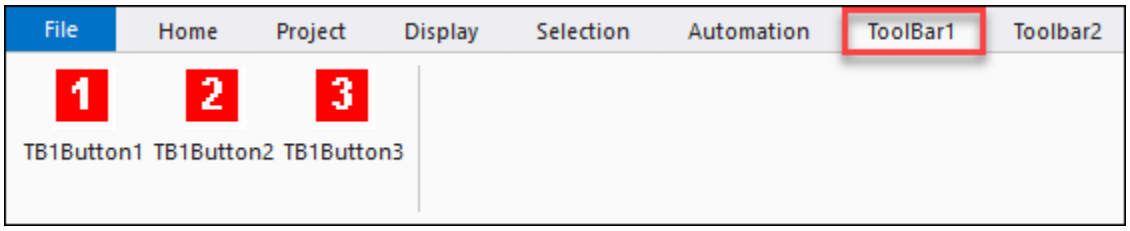
In the introductory example (p. 72), the extension `ExtSample1` shows how to create a custom Mechanical toolbar with a single button that displays this message when clicked: **`High five! ExtSample1 is a success!`**.

Here, the extension `ExtToolbarSample` shows how to create two custom Mechanical toolbars, each with multiple button that call IronPython functions. This extension's XML file follows.

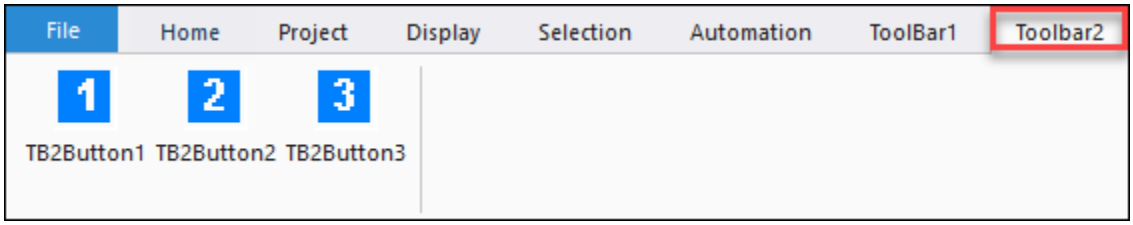
```
<extension version="1" name="ExtToolbarSample">
  <guid shortid="ExtToolbarSample">ccdbbed4-9fdd-4157-acea-abccddbd62fc</guid>
  <script src="toolbarsample.py" />
  <interface context="Mechanical">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
    </callbacks>
    <toolbar name="ToolBar1" caption="ToolBar1">
      <entry name="TB1Button1" icon="button1Red">
        <callbacks>
          <onclick>OnClickTB1Button1</onclick>
        </callbacks>
      </entry>
      <entry name="TB1Button2" icon="button2Red">
        <callbacks>
          <onclick>OnClickTB1Button2</onclick>
        </callbacks>
      </entry>
      <entry name="TB1Button3" icon="button3Red">
        <callbacks>
          <onclick>OnClickTB1Button3</onclick>
        </callbacks>
      </entry>
    </toolbar>
    <toolbar name="Toolbar2" caption="Toolbar2">
      <entry name="TB2Button1" icon="button1Blue">
        <callbacks>
          <onclick>OnClickTB2Button1</onclick>
        </callbacks>
      </entry>
      <entry name="TB2Button2" icon="button2Blue">
        <callbacks>
          <onclick>OnClickTB2Button2</onclick>
        </callbacks>
      </entry>
      <entry name="TB2Button3" icon="button3Blue">
        <callbacks>
          <onclick>OnClickTB2Button3</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
</extension>
```

When the extension `ExtToolbarSample` is loaded in Mechanical, to the right of the ribbon's **Automation** tab, two additional tabs display: **ToolBar1** and **Toolbar 2**. When the **Toolbar1** tab is selected, you see three red buttons:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



When the **Toolbar2** tab is selected, you see three blue buttons:



To define each of these toolbars, the XML file uses the element **`<toolbar>`**. Each of these elements has two attributes, **`name`** and **`caption`**. The attribute **`name`** is required and is used for internal references. The attribute **`caption`** is the text to display in Mechanical.

The element **`<toolbar>`** has a child element **`<entry>`** that defines the buttons in the toolbar. The element **`<entry>`** has two attributes, **`name`** and **`icon`**. The attribute **`name`** is required and is used for internal references. It also displays as the tooltip for the button. The attribute **`icon`** is the name of the image file to display as the button.

The element **`<entry>`** has a child element **`<callbacks>`** that defines callbacks to events. For instance, the callback **`<onclick>`** specifies the name of the IronPython function to invoke when the button is clicked.

Consider the third line in the XML file:

```
<script src="toolbarsample.py" />
```

This line defines `toolbarsample.py` as the IronPython script for the extension. This script follows.

```
import os
import datetime
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *

def init(context):
    ExtAPI.Log.WriteMessage("Init ExtToolbarSample ...")

def OnClickTB1Button1(analysis):
    LogButtonClicked(1, 1, analysis)

def OnClickTB1Button2(analysis):
    LogButtonClicked(1, 2, analysis)

def OnClickTB1Button3(analysis):
    LogButtonClicked(1, 3, analysis)

def OnClickTB2Button1(analysis):
    LogButtonClicked(2, 1, analysis)

def OnClickTB2Button2(analysis):
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
    LogButtonClicked(2, 2, analysis)

def OnClickTB2Button3(analysis):
    LogButtonClicked(2, 3, analysis)

def LogButtonClicked(toolbarId, buttonId, analysis):
    now = datetime.datetime.now()
    outFile = SetUserOutput("ExtToolbarSample.log", analysis)
    f = open(outFile,'a')
    f.write("*.*.*.*.*.*.*.*\n")
    f.write(str(now)+"\n")
    f.write("Toolbar "+toolbarId.ToString()+" - Button "+buttonId.ToString()+" Clicked. \n")
    f.write("*.*.*.*.*.*.*.*\n")
    f.close()
    MessageBox.Show("Toolbar "+toolbarId.ToString()+" - Button "+buttonId.ToString()+" Clicked.")

def SetUserOutput(filename, analysis):
    solverDir = analysis.WorkingDir
    return os.path.join(solverDir,filename)
```

Defining Button Callback Functions

Each button defined in the extension `ExtToolbarSample` has a unique callback function. Each callback function passes the toolbar ID and the ID of the button clicked to the function **`LogButtonClicked`**, which stores them in the variables **`toolbarId`** and **`buttonId`**. These variables are referenced within the function where their string values are written.

The functions **`LogButtonClicked`** and **`SetUserOutput`** demonstrate how to reduce redundant code in callbacks using utility functions. The object **`Analysis`** is passed to each callback **`<entry>`** and then used in the function **`SetUserOutput`** to query for the working directory of the analysis.

The script in `toolbarsample.py` makes use of the namespace **`datetime`** from the .NET framework. The namespace **`datetime`** exposes a class that is also called **`datetime`**. The function **`LogButtonClicked`** invokes **`datetime`** to query the current date and time and then stores the result in the variable named **`now`**. The utility **`str()`** is used to extract the string definition of the variable **`now`** to write out the date and time.

Binding Toolbar Buttons with ACT Objects

ACT provides the ability to bind a button from the ACT toolbar with an ACT object created in the Mechanical tree. This capability allows you to control the contextual availability of the buttons depending on the object selected in the tree. This method is similar to the control used for standard objects in Mechanical. To make the connection between the ACT object and the ACT button, in the XML file, you use the attribute **`userobject`** in the child element **`<entry>`** for the element **`<interface>`**. The following code demonstrates this type of connection for a result object:

```
<interface context="Mechanical">
  <images>images</images>
  <toolbar name="My_Toolbar" caption="My_Toolbar">
    <entry name="Button_For_My_Result icon="result" userobject="My_Result">
    </entry>
  </toolbar>
</interface>

<simdata context="Mechanical">
  <result name="My_Result" version="1" caption="My_Result" icon="result" location="elemnode"
type="scalar">
  </result>
</simdata>
```

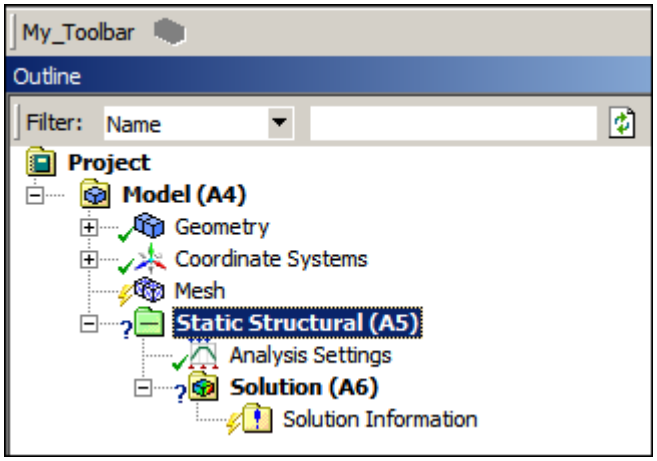

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

As an example, if the ACT button is bound with an ACT load in Mechanical, this button is activated only if the object is selected in the Mechanical environment. Otherwise, it is inactive.

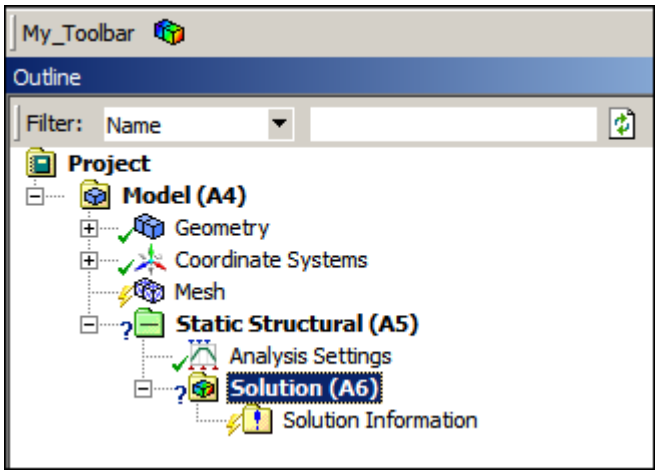
In the same way, if the ACT button is bound with an ACT result in Mechanical, this button is active only if the object is selected in the solution. Otherwise, it is inactive.

For any ACT object bound to a button, the callback **<onclick>** is not used.

The following figure illustrates behavior based on selection of an environment object in the tree. The button is inactive.



This next figure illustrate the behavior based on selection of a solution object in the tree. The button is active.



In addition to the control provided by the connection between the ACT button and the ACT object, the callback **<canadd>** can be implemented to add new criteria to consider for the activation and deactivation of the button. If the callback **<canadd>** of the object returns **false**, the associated button is deactivated. A typically example consists of filtering a particular analysis type to activate a specific load.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Dialog Box Creation

You can add custom dialog boxes to Ansys products. The extension `ExtDialogSample` defines a custom menu in Mechanical named **DialogSample** that has one menu item named **GetFilename**.This menu option opens a file selection dialog box and displays a custom message dialog box. The XML file follows.

```
extension version="1" name="ExtDialogSample">
  <guid shortid="ExtDialogSample">25c00857-4e27-4b01-bd22-c52699ac39e9</guid>
  <script src="dialogsample.py" />
  <interface context="Mechanical">
    <images>images</images>
    <callbacks>
      <oninit>init</oninit>
    </callbacks>
    <toolbar name="DialogSample" caption="DialogSample">
      <entry name="DialogSample1" caption="GetFilename" icon="blank">
        <callbacks>
          <onclick>GUIMenuOpenFile</onclick>
        </callbacks>
      </entry>
    </toolbar>
  </interface>
</extension>
```

The callback function specified for the **GetFilename** menu button is **`GUIMenuOpenFile`**. The IronPython script `dialogsample.py` defines this function.

```
clr.AddReference("Ans.UI.Toolkit")
clr.AddReference("Ans.UI.Toolkit.Base")
from Ansys.UI.Toolkit import *

def init(context):
    ExtAPI.Log.WriteMessage("Init ExtDialogSample ...")

def GUIMenuOpenFile(analysis):
    filters = "txt files (*.txt)|*.txt|All files (*.*)|*.*"
    dir = "c:\\"
    res = FileDialog.ShowOpenDialog(ExtAPI.UserInterface.MainWindow,dir,filters,2,"ExtDialogSample","")

    if res[0]==DialogResult.OK:
        message = str("OPEN selected -> Filename is "+res[1])
    else:
        message = "CANCEL selected -> No file"
    MessageBox.Show(message)
```

When the function **`GUIMenuOpenFile`** is invoked, an instance of a modal file selection dialog box is created by calling **`FileDialog.ShowOpenDialog`**. The class **`FileDialog`** is provided by the UI Toolkit. When running the extension, the necessary information is supplied in the dialog box. When the **Open** or **Cancel** button is clicked, the file selection dialog box closes. The supplied information is returned to the function **`GUIMenuOpenFile`**, which uses it to create the message dialog box. The message shown in this dialog box validates the result that is returned from the file selection dialog box.

Extension Data Storage

ACT provides two mechanisms for storing data in your extension. These mechanisms are based on the callbacks **`<onload>`** and **`<onsave>`** or on attributes.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- The callback **<onsave>** is called each time the target product saves the project. Consequently, this callback allows the creation of dedicated files in which to store data. It also allows data to be stored in the standard Ansys Workbench project.
- The callback **<onload>** is called each time the target product loads the project. The process here is similar to the callback **<onsave>**, but it is now devoted to reading in additional data.
- Attributes represent an interesting way to store data because they do not require the creation of external files. The remainder of this describes how to use attributes to store data.

Attributes are defined by name and content. The content can be defined by a single value, such as an integer or a double, or it can be defined with more complex data. The content must be serializable. To accomplish this, implement the serialization interface provided by the .NET framework. Types such as **integer**, **double**, **list**, and **dictionary** are serializable by default.

Attributes are automatically saved and resumed with the project. Attributes can be associated with an ACT extension, load or result, or property.

Attributes are created or edited with the method:

```
Attributes["attribute_name"] = attribute_value
```

Content can be retrieved with the method:

```
attribute_value = Attributes["attribute_name"]
```

An example follows of an attribute associated with a property:

```
prop.Attributes["MyData"] = 2
val = prop.Attributes["MyData"]
```

A similar example follows for an attribute associated with a load:

```
load.Attributes["MyData"] = 2
val = load.Attributes["MyData"]
```

A similar example follows for an attribute associated with an extension:

```
ExtAPI.ExtensionMgr.CurrentExtension.Attributes["MyData"] = 2
v = ExtAPI.ExtensionMgr.CurrentExtension.Attributes["MyData"]
```

An attribute associated with an extension can be shared between products using the same extension. For example, two Mechanical sessions can share data.

The command to store the attribute in a shared repository is:

```
ExtAPI.ExtensionMgr.CurrentExtension.SetAttributeValueWithSync("MyData", 2.)
```

Similarly, the command to retrieve the content stored in a shared repository is:

```
ExtAPI.ExtensionMgr.CurrentExtension.UpdateAttributes()
v = ExtAPI.ExtensionMgr.CurrentExtension.Attributes["MyData"]
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

ACT-Based Property Creation

ACT has the ability to create customized objects that encapsulate ACT-based properties. Two methods exist for using ACT to create property groups and properties:

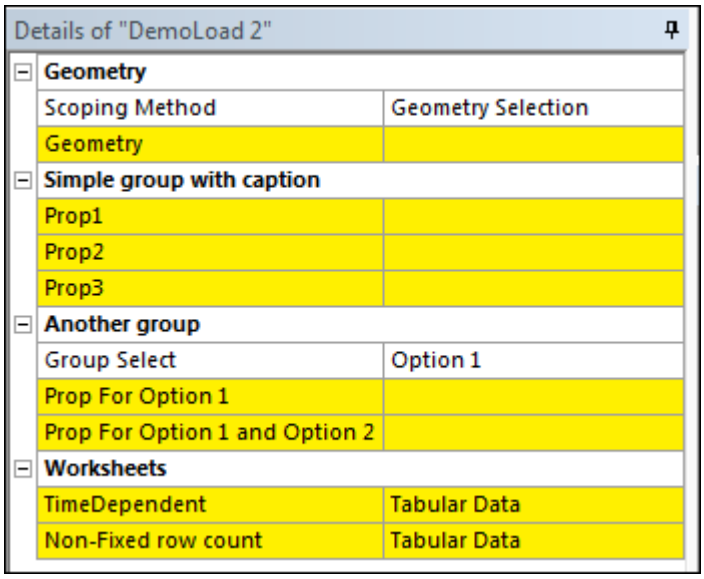
Using the Elements <PropertyGroup> and <PropertyTable> for Property Creation

This topic describes how to create groups of properties using the elements **<PropertyGroup>** and **<PropertyTable>**. In the XML extension definition file, you use these two elements to create groups of properties with a given caption. In this way, it is possible to manage dependencies between properties and to create worksheet views from a property.

- The element **<PropertyGroup>** encapsulates a list of child properties under one group.
- The element **<PropertyTable>** encapsulates a list of child properties under one table. Each child property creates a new column in the table. You can control the line number of this table.

These extension `AdvancedProperties` demonstrates how to use the elements **<PropertyGroup>** and **<PropertyTable>**.

Assume that you want to create a group of properties with a caption and that you want to be able to collapse and expand this group. The following figure shows the caption **Simple group with caption**.



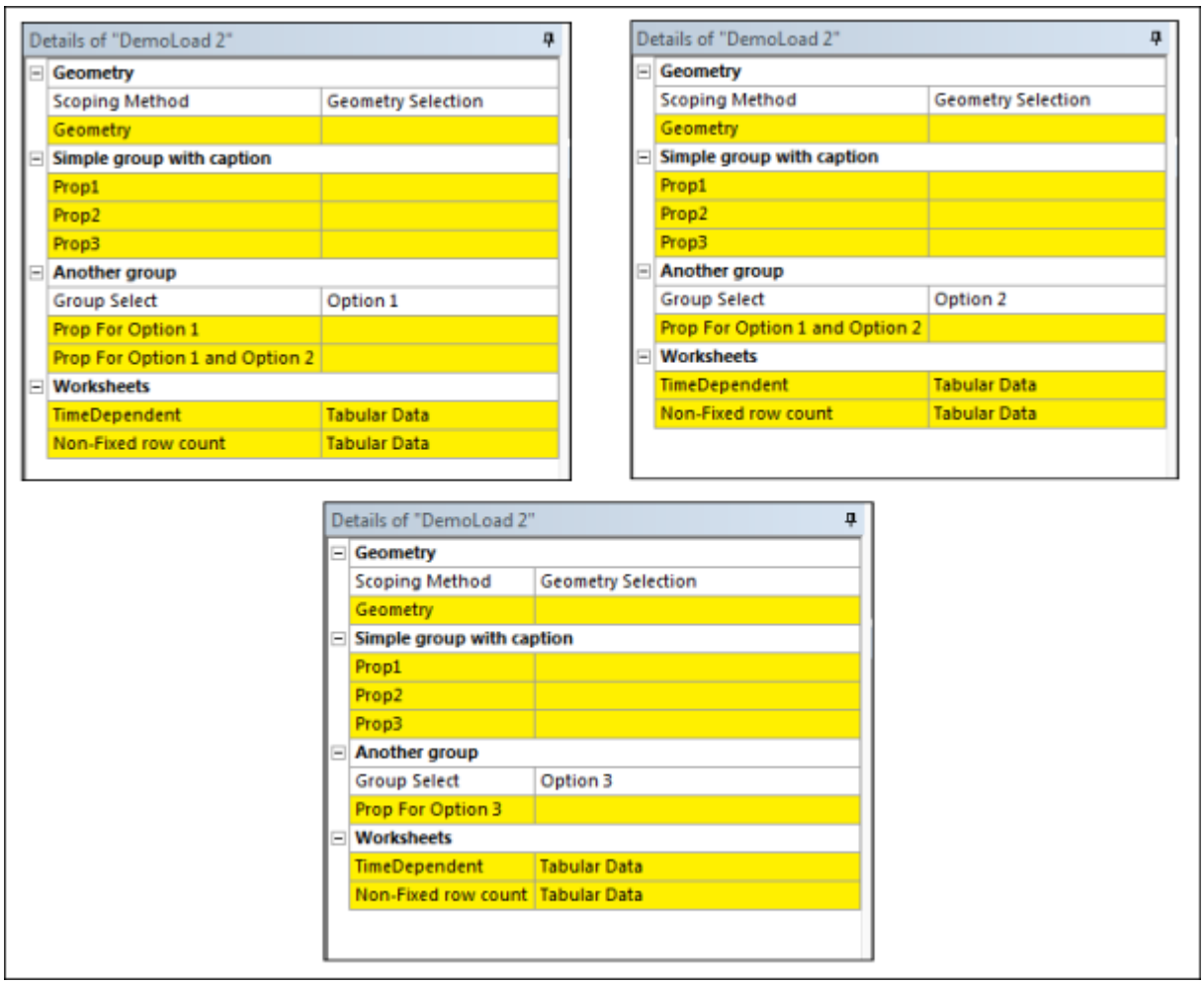
This group is created with the following XML code:

```
<propertygroup name="Group1" caption="Simple group with caption" display="caption">
  <property name="Prop1" caption="Prop1" control="text" />
  <property name="Prop2" caption="Prop2" control="text" />
  <property name="Prop3" caption="Prop3" control="text" />
</propertygroup>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

The element **propertygroup** has a special attribute, **display**. In this case, **display** is set to **caption**, which indicates that all child properties are to display under the caption. If **caption** is omitted, **display** defaults to **hidden**, which indicates that the property group is hidden.

Assume now that you want to show or hide properties according to the value of another selected property. In the following figure, the visibility of the properties depends on the value of the property **Group Select**.



This group is created with the following XML code:

```
<propertygroup name="Group2" caption="Another group" display="caption">
  <propertygroup name="Group3" caption="Group Select" display="property" control="select" default="Option 1">
    <attributes options="Option 1,Option 2,Option 3" />
    <property name="Prop1" caption="Prop For Option 1" visibleon="Option 1" control="text" />
    <property name="Prop2" caption="Prop For Option 1 and Option 2"
     visibleon="Option 1|Option 2" control="text" />
    <property name="Prop3" caption="Prop For Option 3" visibleon="Option 3" control="text" />
  </propertygroup>
</propertygroup>
```

In this case, the attribute **display** is set to **property**. The element **propertygroup** named **Group3** defines a standard ACT property that provides additional capabilities for all child properties.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

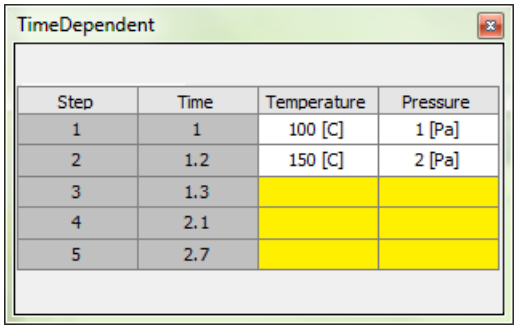
Each child property can specify an attribute **visibleon**, which can take a value or a set of values. If the current value of the parent property fits with the attribute **visibleon**, the property is displayed. Otherwise, the property is hidden.

You can also create properties that open a worksheet in which to access the content for properties. You use the type **PropertyTable** if you want to create a worksheet that exposes a set of properties for your customization. To facilitate development, ACT provides two different types of predefined worksheets:

- Time-dependent
- Non-fixed row dimension

Time-Dependent Worksheets

In a time-dependent worksheet, the row number is initialized with the number of steps defined in the object **AnalysisSettings**. When time steps are added to or removed from the object **AnalysisSettings**, the worksheet is automatically updated. Consequently,this type of worksheet represents an efficient way to manage time-dependent data within an extension.



The following XML code creates a time-dependent worksheet:

```
<propertytable name="TimeDep" caption="TimeDependent" display="worksheet" control="applycancel" class="Worksheet
  <property name="Step" caption="Step" control="integer" readonly="true" />
  <property name="Time" caption="Time" control="float" readonly="true" />
  <property name="Temperature" caption="Temperature" unit="Temperature" control="float"></property>
  <property name="Pressure" caption="Pressure" unit="Pressure" control="float"></property>
</propertytable>
```

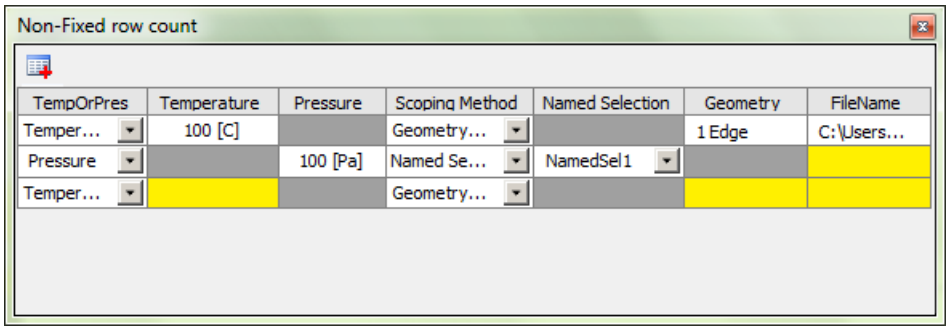
In this example, the attribute **display** is set to **worksheet**. In addition, the attribute **class** specifies the name of the IronPython class that manages the worksheet. This example uses the class **TFTabularData**, which is defined in the function library `TimeFreqTabularData.py`. This library (p. 115) is installed with ACT and located in `%ANSYSversion_DIR%\Addins\ACT\libraries\Mechanical\Worksheet`.

The properties **Step** and **Time** integrate a specific treatment, as they are automatically populated with the information specified in the object **AnalysisSettings**. These two properties are optional.

Non-Fixed Row Dimension Worksheets

In a non-fixed row dimension worksheet, the array is empty by default. You can add a new line by clicking the top left button.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



The following XML code creates a non-fixed row dimension worksheet:

```
<propertytable name="Worksheet" caption="Non-Fixed row count" display="worksheet" control="applycancel" class="w
  <propertygroup name="TempOrPres" caption="TempOrPres" display="property" control="select" default="Temperature
    <attributes options="Temperature,Pressure" />
    <property name="Temperature" caption="Temperature" unit="Temperature" control="float"
     visibleon="Temperature"></property>
    <property name="Pressure" caption="Pressure" unit="Pressure" control="float" visibleon="Pressure"></property
  </propertygroup>

  <property name="Scoping" caption="Scoping" control="scoping">
    <attributes selection_filter="face|edge" />
  </property>
  <property name="FileName" caption="FileName" control="fileopen">
    <attributes filters="Command files (*.bat)|*.bat|All files (*.*)|*.*" />
  </property>
</propertytable>
```

This example uses the class **`PGEditor`**, which is defined in the function library `PropertyGroupEditor.py`. This library (p. 115) is installed with ACT and located in `%ANSYSversion_DIR%\Addins\ACT\libraries\Mechanical\Worksheet`.

You access the content of the worksheet in the same manner as you do any other standard ACT property.

Using Templates for Property Creation

A template represents a generic method for defining a group of properties for an associated callback. You can create a template to provide services that can be integrated into any extension. For example, you can build a template to generate a property that displays all available coordinate systems for the current model, allowing you to select between them. You can also enrich the templates currently integrated in ACT to effectively customize environments.

The directory *`ANSYS_INSTALL_DIR`*`\Addins\ACT\templates` contains folders for various Ansys products. Each folder contains a single XML file with one or more templates, each of which is assigned a name. For example, the folder `Mechanical` contains the file `controltemplates.xml`, which contains templates for scoping, opening a file, and selecting a component, geometry, entity, coordinate system, and custom unit.

The template `coordinatesystem_selection` follows. It uses the class **`SelectCoordinateSystem`**, which is defined in the file `select.py`. This Python file is located here: *`ANSYS_INSTALL_DIR`*`\Addins\ACT\libraries\Mechanical\templates`. To link a template to a property, in its attributes, you set **`control`** to the name of the template.

```
</controltemplate>
    <!-- Coordinate System Selection -->
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
-<controltemplate version="2" name="coordinatesystem_selection">
    <property name="selectCoordinateSystem" class="templates.select.SelectCoordinateSystem" control="select"/>
</controltemplate>
```

Note:

A template must be made of one single property. If several properties are to be defined, you must integrate them into a group.

For more information about the templates for an Ansys product, go to its folder in *ANSYS_INSTALL_DIR*\Addins\ACT\templates and open its XML file. For example, for more information about Mechanical templates, go to *ANSYS_INSTALL_DIR*\Addins\ACT\templates\Mechanical and open controltemplates.xml.

ACT-Based Property Parameterization

All ACT-based properties are treated as input parameters unless you specify otherwise in the extension's XML file. You can also specify whether to treat ACT parameters as input or output parameters directly in DesignModeler, Mechanical, and Workbench.

In your analyses, you can incorporate parameters in a variety of places.

- In Design Modeler, you can incorporate parameters into a number of custom ACT-based features, such as renaming selected bodies and specifying geometry generation information.
- In Mechanical, you can define parameters for any ACT object, regardless of its location in the tree.
- In Workbench, you can define parameters for custom task-level properties in the **Project Schematic**.
- In a third-party solver implemented by ACT, you can incorporate parameters into the load, analysis settings, and result.

Once parameterized, an ACT property is added to the **Parameter Set** bar in the Workbench **Project Schematic**, where it behaves and is treated as any other parameter.

All combinations of ACT-based and standard parameters are supported:

- ACT inputs and standard outputs
- ACT inputs and ACT outputs
- Standard inputs and ACT outputs

The following topics provide general information on parametrizing ACT properties in extensions and for third-party solvers:

Parametrizing ACT-Based Properties in Extensions

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Parametrizing ACT-Based Properties for Third-Party Solvers

Note:

Product-specific information on parametrizing ACT-based properties in extensions and from third-party solvers is available in the ACT customization guides for particular products. For links to the DesignModeler, Mechanical, and Workbench guides, see ACT Customization Guides for Supported Ansys Products (p. 183).

Parametrizing ACT-Based Properties in Extensions

To define an ACT property as a parameter in the extension's XML, you add the attribute **`isparameter`** and set it to **`true`**. By default, an ACT property is created as an input parameter. To define it as an output parameter, you set the attribute **`readonly`** to **`true`**.

When you define a property as a parameter, it is not parameterized by default. Defining a property as a parameter only makes parameterization possible by adding a check box to the Ansys product. To actually parameterize the property, you must select the check box that the attribute **`isparameter`** makes available in the product.

Once you select a parameter for parameterization, it is automatically sent to the **Parameter Set** bar in the Workbench **Project Schematic**. You can view it in both the **Outline** and **Table** panes for the **Parameter Set** bar. While output parameters are read-only, you can set input parameters from ACT in the same way as any other input parameter. For example, you can:

- Change the unit of the parameterized input property. The unit proposed respects the type of unit specified in the XML file.
- Add a design point for the input parameter by clicking in the bottom row of the design points table.
- Modify the value of a design point by selecting an input parameter and entering a new value. However, the type of value must respect the one specified in the XML file. As such, design point values in the previous value must be floats. They cannot be strings.

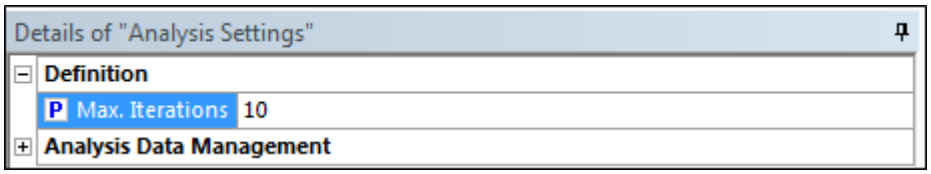
There should be at least one input parameter and one output parameter. When you finish setting up the system and working with parameters and design point values, you can solve the problem by updating the design points for the **Parameter Set** bar.

Parametrizing ACT-Based Properties for Third-Party Solvers

When you use ACT to deploy a third-party solver, you can define the solver's ACT-based properties as either input parameters or output parameters. Parametrizing ACT-based properties for a third-party solver is no different than parametrizing ACT properties in the extension. In the property definition, you add the attribute **`isparameter`** and set it to **`true`**. By default, the property is an input parameter. To define it as an output parameter, you set the attribute **`readonly`** to **`true`**.

You can also parameterize analysis settings for a third-party external solver. The settings available depend on the definition of the third-party solver. For example, you can parameterize the maximum number of iterations:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



In the XML file, the element **<solver>** defines the property **MaxIter**. For this property, you set the attribute **isparameter** to **true**.

```
<solver…

    <property name="MaxIter" caption="Max. Iterations" control="integer" isparameter="true" default="10"/>
</solver>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Simulation Wizards

You can use ACT to create *simulation wizards* for all Ansys products listed in the introduction (p. 1). A wizard automates a simulation process by walking non-expert users step-by-step through the simulation.

A wizard is part of an ACT extension (p. 71). Within the element **<extension>**, you add the element **<wizard>** and then define each step, along with step callbacks and properties. A wizard can access the functionality defined in the wizard and leverage API scripting capabilities, which allows you to benefit from Ansys simulation capabilities while minimizing interactions with Ansys products.

The following topics are addressed:

Wizard Interface and Usage

Wizard Types

Wizard Creation

Mixed Wizard Example

Custom Wizard Help Files

Custom Wizard Interfaces

Custom Wizard Interface Example

Note:

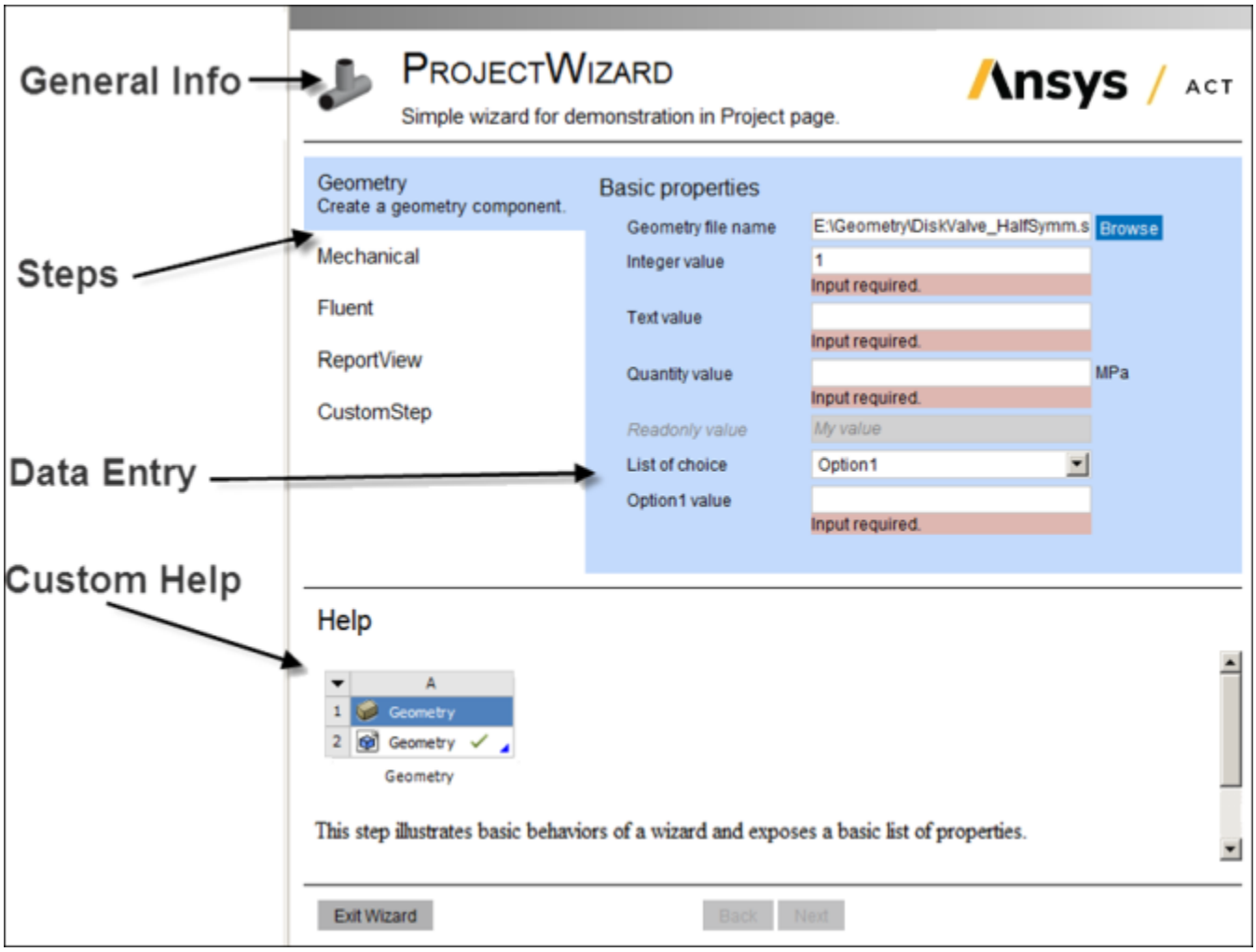
You use the Extension Manager (p. 41) to install and load extensions. You then use the Wizards launcher (p. 47) to start a wizard.

Wizard Interface and Usage

The default interface for a Workbench-based wizard is divided into four panels:

- The top panel shows general information about the wizard and a logo of your choice.
- The left panel shows a list of the steps in the wizard, with the current step highlighted.
- The right panel shows data-entry and data-display fields for the selected step.
- The bottom panel shows custom help for the selected step.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



Note:

The overall content and function of a wizard is consistent across Ansys products. For wizard information specific to a particular product, see the ACT customization guide for the product. The concluding section (p. 183) provides links to these guides.

Entering Data in a Wizard

For each step in a wizard, you enter data, which is then validated as specified in the extension. When all required values are entered and pass validation, the **Next** button becomes enabled. When you click this button, the data is submitted and the action defined in the step is executed.

The wizard definition can include a progress bar to display progress as the step executes and a message box to display a confirmation when the step execution completes successfully. In the **Project** tab, you can verify completion of the step.

Once you are past the first step, the **Back** button is enabled. When you click this button, the wizard returns to the previous step, but the data generated for this step is lost.

To enter tabular data in a wizard, you do the following:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

1. Click **Edit**.
2. Click **Add** to add a table row.
3. Enter values into the cells.
4. When finished, click either the green check box icon to save the data or the red stop icon to cancel.

Exiting a Wizard

On the final step of a wizard, the **Finish** button is enabled. When you click this button, the wizard completes. You are returned to the **ACT Start Page**.

To exit a wizard at any time, you can click **Exit Wizard** in the lower left corner of the wizard interface. You are returned to the **Wizards** launcher.

Note:

If you exit the wizard midway through, the changes to the project are retained and can be saved, but the data that you entered in the wizard is not retained. You cannot resume the wizard where you left off.

To exit the project with a wizard still in progress, save the project and exit as usual. On reopening the project, the wizard is still in its last saved state, so you can resume where you left off.

Wizard Types

You can create three different types of wizards: project wizards, target product wizards, and mixed wizards.

Tip:

The extensions referenced in this section are supplied. For download information, see Extension and Template Examples (p. 81).

Project Wizards

A *project wizard* is executed from the **Project** tab in Workbench. A project wizard can engage any data-integrated Ansys product with Workbench journaling and scripting capabilities and incorporate them into the project workflow. The extension **`DemoWizards`** includes a project wizard named **`ProjectWizard`**. This wizard is described in Wizard Creation (p. 144).

Target Product Wizards

A *target product wizard* is executed wholly in a specified Ansys product that provides both scripting capabilities and ACT support. A target product wizard utilizes the functionality provided by the target product and provides simulation guidance within the product's workflow.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Target product wizards for some products, such as native products like DesignModeler and Mechanical, execute only within the Workbench environment. Target product wizards for other products, such as Electronics Desktop, Fluent, and SpaceClaim, can execute either within the Workbench environment or in a stand-alone instance of the product.

The following table shows where different target product wizards can be executed. For product-specific wizard examples, see the ACT customization guides for these products. The guides for Electronics Desktop, Fluent, and SpaceClaim also provide usage information for a stand-alone instance of the product. The concluding section (p. 183) provides links to these guides.

Ansys Product	Executed in Workbench Environment	Executed in Stand-alone Instance of the Product
DesignModeler	X	
Electronics Desktop	X	X
Fluent	X	X
Mechanical	X	
SpaceClaim	X	X

Mixed Wizards

A *mixed wizard* is executed from the Workbench **Project** tab and engages one or more supported Ansys products to automate the entire simulation process. A mixed wizard is used when interactive selection on the model is required at a given step in one given product. It provides native workflow guidance in both the **Project** tab and the included products.

The extension **`DemoWizards`** includes a mixed wizard named **`BridgeSimulation`**, which is described in Mixed Wizard Example (p. 155).

Note:

Because DesignXplorer is part of Workbench, you can create project wizards and mixed wizards that can engage DesignXplorer. For the wizard step, you simply need to set the attribute **`<context>`** to **`Project`**. Because DesignXplorer does not have a stand-alone application interface, you cannot create target product wizards for DesignXplorer.

Wizard Creation

To create a wizard, in the extension's XML file, you include specific elements that define the wizard. If desired, you can create HTML files to supply custom wizard help (p. 160).

The following topics describe how to create a project wizard, using the supplied extension **`WizardDemos`** as an example:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Adding the XML Element for the Wizard

An extension can contain any number of wizards. For each wizard that you want to create, you add the element **<wizard>** in the extension's XML file. For each wizard, you define steps. For each step, you define callbacks and properties.

Tip:

To customize the appearance of a Workbench-based wizard, you can add the optional element **<uidefinition>**. In this element, you can define layouts both for the wizard interface as a whole and for individual wizard components. For more information, see Custom Wizard Interfaces (p. 162) and Custom Wizard Interface Example (p. 164).

The extension **WizardDemos** contains a project wizard named **ProjectWizard** that is executed from the **Project** tab in Workbench. An excerpt from the file **WizardDemos.xml** follows. Code is omitted for the element **<uidefinition>** and all wizards other than the wizard **ProjectWizard**.

```
<extension version="2" minorversion="1" name="WizardDemos">
 <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
 <author>Ansys, Inc.</author>
 <description>Simple extension to test wizards in different contexts.</description>

 <script src="main.py" />
 <script src="ds.py" />
 <script src="dm.py" />
 <script src="sc.py" />

 <interface context="Project|Mechanical|SpaceClaim">
  <images>images</images>
 </interface>

 <interface context="DesignModeler">
  <images>images</images>

  <toolbar name="Deck" caption="Deck">
   <entry name="Deck" icon="deck">
    <callbacks>
     <onclick>CreateDeck</onclick>
    </callbacks>
   </entry>
   <entry name="Support" icon="Support">
    <callbacks>
     <onclick>CreateSupport</onclick>
    </callbacks>
   </entry>
  </toolbar>

 </interface>

 ...

 <wizard name="ProjectWizard" version="1" context="Project" icon="wizard_icon">
  <description>Simple wizard for demonstration in Project page.</description>

  <step name="Geometry" caption="Geometry" version="1" HelpFile="help/geometry.html" layout="DefaultTabularDataL
   <description>Create a geometry component.</description>

   <componentStyle component="Submit">
    <background-color>#b6d7a8</background-color>
   </componentStyle>

   <componentData component="TabularData">
```

```
    <CanAddDelete>false</CanAddDelete>
   </componentData>

   <callbacks>
    <onupdate>EmptyAction</onupdate>
    <onreset>DeleteGeometry</onreset>
    <oninit>InitTabularData</oninit>
   </callbacks>

   <propertygroup display="caption" name="definition" caption="Basic properties" >
    <property name="filename" caption="Geometry file name" control="fileopen" />
    <property name="myint" caption="Integer value" control="integer" />
    <property name="mytext" caption="Text value" control="text" />
    <property name="myquantity" caption="Quantity value" control="float" unit="Pressure" />
    <property name="myreadonly" caption="Readonly value" control="text" readonly="true" default="My value" />
    <propertygroup display="property" name="myselect" caption="List of choice" control="select" default="Option1
     <attributes options="Option1,Option2" />
     <property name="option1" caption="Option1 value" control="text" visibleon="Option1" />
     <property name="option2first" caption="Option2 first value" control="float" unit="Pressure" visibleon="Opti
     <property name="option2seond" caption="Option2 second value" control="float" unit="Length" visibleon="Optio
    </propertygroup>
   </propertygroup>

   <propertytable name="Table" caption="Table" control="custom" display="worksheet" class="Worksheet.TabularData
    <property name="Frequency" caption="Frequency" unit="Frequency" control="float" isparameter="true"></propert
    <property name="Damping" caption="Damping" control="float" isparameter="true"></property>
    <property name="TestFileopen" caption="fileopen" control="fileopen"></property>
   </propertytable>

   <propertytable name="TableB" caption="Table B" control="tabulardata" display="worksheet">
    <property name="Integer" caption="Integer" control="integer"></property>
    <property name="FileOpen" caption="Fileopen" control="fileopen"></property>
    <property name="Number" caption="Number A" unit="Pressure" control="float"></property>
    <property name="ReadOnly" caption="ReadOnly" unit="Pressure" control="float" readonly="true" default="4 [Pa]
    <propertygroup display="property" name="myselect" caption="List of choice" control="select" default="Option1
     <attributes options="Option1,Option2" />
     <property name="TestStr" caption="Text" control="text" visibleon="Option1"></property>
     <property name="Number B" caption="Number B" unit="Temperature" control="float"></property>
    </propertygroup>
   </propertytable>

  </step>

  <step name="Mechanical" caption="Mechanical" enabled="true" version="1" HelpFile="help/mechanical.html">
   <description>Create a mechanical component.</description>

   <callbacks>
    <onupdate>EmptyAction</onupdate>
    <onreset>DeleteMechanical</onreset>
   </callbacks>

   <property name="name" caption="System name" control="text" />

  </step>

  <step name="Fluent" caption="Fluent" version="1" HelpFile="help/fluent.html">
   <description>Create a fluent component.</description>

   <callbacks>
    <onrefresh>CreateDialog</onrefresh>
    <onupdate>EmptyAction</onupdate>
    <onreset>EmptyReset</onreset>
   </callbacks>


   <property name="name" caption="System name" control="text" />
   <property name="dialog" caption="Dialog" control="text">
    <callbacks>
     <onvalidate>ValidateDialog</onvalidate>
    </callbacks>
   </property>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
   <property name="dialog2" caption="DialogProgress" control="text">
    <callbacks>
     <onvalidate>ValidateDialogProgress</onvalidate>
    </callbacks>
   </property>
   <property name="nextstep" caption="Next Step" control="select" >
    <callbacks>
     <onvalidate>ValidateNextStep</onvalidate>
    </callbacks>
   </property>

  </step>

  <step name="ReportView" caption="ReportView" version="1" layout="ReportView@WizardDemos">
   <description>Simple example to demonstrate how report can be displayed.</description>

   <callbacks>
    <onrefresh>RefreshReport</onrefresh>
    <onreset>EmptyReset</onreset>
   </callbacks>

  </step>

  <step name="CustomStep" caption="CustomStep" enabled="true" version="1" layout="MyLayout@WizardDemos">
   <description>Create a mechanical component.</description>

   <callbacks>
    <onrefresh>RefreshMechanical</onrefresh>
    <onupdate>LogReport</onupdate>
    <onreset>EmptyReset</onreset>
   </callbacks>

   <property name="name" caption="System name" control="text" />

  </step>

  <step name="Charts" caption="Charts" enabled="true" version="1" layout="GraphLayout@WizardDemos">
   <description>Demonstrate all chart capabilities.</description>

   <callbacks>
    <onrefresh>RefreshCharts</onrefresh>
   </callbacks>

  </step>

 </wizard>

 ...

</extension>
```

As indicated in the element **<description>**, this extension is for testing wizards in different contexts. It has two elements **<interface>**. In the first element **<interface>**, the attribute **context** is set to **Project|Mechanical|SpaceClaim**, which indicates that wizards with their context set to these products use this interface. In the second element **<interface>**, the attribute **context** is set to **DesignModeler**, which indicates that wizards with their context set to this product use this interface.

In addition to the project wizard named **ProjectWizard**, the extension **WizardDemos** includes target product wizards for DesignModeler, SpaceClaim, and Mechanical. Additionally, it includes a mixed wizard (p. 155). For information on the product-specific wizards in this extension, see the ACT customization guides for these products. The concluding section (p. 183) provides links to these guides.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Defining the Wizard

For the element **<wizard>**, the required attributes are **name**, **version**, and **context**. In this wizard example, the attribute **name** is set to **ProjectWizard**. The attribute **<context>** is set to **Project**.

The optional attributes are **icon** and **description**.

- The attribute **icon** specifies the filename of the image to display as the icon for the wizard: `wizard_icon`. Any image file specified for use as an icon must be placed in the folder declared for images in the element **<interface>** for the extension. The child element **<images>** specifies the folder name.
- The attribute **description** specifies the text to display in the **About** window for the extension when it is accessed from the **Extension Manager** and in the **About** window for the wizard when it is accessed from the **Wizards** page.

Note:

For a mixed wizard, the attribute **context** is set to **Project**. A mixed wizard can only be executed from the **Project** tab in Workbench.

Step Definition

You use the element **<step>** to define a step in the wizard. The wizard **ProductWizard** has six steps: **Geometry**, **Mechanical**, **Fluent**, **ReportView**, **CustomStep**, and **Charts**.

Required attributes for the element **<step>** are **name** and **version**. The optional attribute **caption** sets the display text for the step.

You only need to set the attribute **context** if the context for the step differs from the context for the wizard. Because all six steps in the wizard **ProjectWizard** are executed from the Workbench **Project** tab, the attribute **context** does not have to be set.

Note:

For a mixed wizard, the attribute **context** is set to the Ansys product from which the step is executed.

Additional optional attributes can be defined for steps. For example, the attribute **HelpFile** can specify an HTML file with custom help (p. 160) to display for the step.

The attribute **enabled** specifies whether to show or hide the step. This attribute is set to true by **default**.

In the IronPython script, you can programmatically change the state of the attribute **enabled** to create branches where certain steps are executed in only specific circumstances: **step.Enabled = True** or **step.Enabled = False**.

Consequently, when the wizard is executed, the choices made in the step determine which subsequent steps display.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Callback Definition

Each step defined in the XML file can have multiple callbacks invoking functions defined in the IronPython script. You use the element **<callbacks>** to define the callbacks for a step. Each callback receives the current step as a method argument.

- The required callback **<onupdate>** is invoked when **Next** is clicked to move to the next wizard step.
- The callback **<onrefresh>** is invoked when the next step is initialized. It initiates changes to the interface, so you can use this callback to change the appearance of the wizard, initialize a value to use in the current step, instantiate variables, access data of an existing project, and so on.
- The callback **<onreset>** resets the state of the step so that a step completed incorrectly can be corrected.

A summary follows of the callbacks for the steps in the wizard **ProductWizard**:

- For the step **Geometry**, the callback **<onupdate>** executes the function **EmptyAction**. The callback **<onreset>** executes the function **DeleteGeometry**. The callback **<oninit>** executes the function **InitTabularData**.
- For the step **Mechanical**, the callback **<onupdate>** executes the function **EmptyAction**. The callback **<onreset>** executes the function **DeleteMechanical**.
- For the step **Fluent**, the callback **<onrefresh>** executes the function **CreateDialog**. The callback **<onupdate>** executes the function **EmptyAction**. The callback **<onreset>** executes the function **EmptyReset**.
- For the step **ReportView**, the callback **<onrefresh>** executes the function **RefreshReport**. The callback **<onreset>** executes the function **EmptyReset**.
- For the step **CustomStep**, the callback **<onrefresh>** executes the function **RefreshMechanical**. The callback **<onupdate>** executes the function **LogReport**. The callback **<onreset>** executes the function **EmptyReset**.
- For the step **Charts**, the callback **<onrefresh>** executes the function **RefreshCharts**.

Functions can also be executed by callbacks within property definitions for steps.

Tip:

In IronPython scripts, functions can use Ansys journaling and scripting commands. You can paste commands obtained from a session journal file—either from the temporary journal stored in your folder **%TEMP%** or from a journal generated by using the **Record Journal** menu option—and then edit these commands.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Property Definition

You use the element **<propertygroup>** to define a group of properties within a step. You use the element **<property>** to define a property and its attributes. For a property requiring validation, the callback **<onvalidate>** executes the appropriate validation function.

For the element **<property>**, required attributes are **control** and **name**. The attribute **control** specifies the property type. Standard property types are **select**, **float**, **integer**, **scoping**, **text**, and **fileopen**. Advanced property types such as **applycancel** and **custom** are also supported.

Optional attributes can be defined for various properties.

- The attribute **caption** specifies the display text for the property.
- The attribute **unit** specifies the units for a property value.
- The attribute **readonly** specifies whether a property value can be edited.
- The attribute **default** specifies a default property value.
- The attribute **visibleon** specifies whether a property is to display. In this example, the property **visibleon** is used to control the display of conditional options in a property group.

In the wizard **ProjectWizard**, the step **Geometry** includes multiple occurrences of the element **<propertygroup>**. The first is named **definition**. Because the attribute **control** is not defined, this property group lists properties individually. Embedded with this property group is a second property group named **myselect**. It sets the attribute **control** to **select**, which provides a drop-down list:

- The attribute **options** defines the available options.
- The attribute **visibleon** indicates that the properties in the group are conditional based on selection from the drop-down.

The step **Geometry** also includes the element **<propertytable>**, which creates a table for the entry of property values in tabular format. The properties **Temperature** and **Pressure** are nested inside the element **<propertytable>** to create columns for data entry. For information on entering tabular data, see Entering Data in a Wizard (p. 142).

Defining Functions for the Wizard

The XML file for the extension **WizardDemos** references multiple IronPython scripts:

```
<script src="main.py" />
 <script src="ds.py" />
 <script src="dm.py" />
 <script src="sc.py" />
```

The script `main.py` defines the functions executed by the callbacks for steps in the wizard **ProjectWizard**.

```
geoSystem = None
dsSystem = None
fluentSystem = None
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
def EmptyAction(step):
    pass

def InitTabularData(step):
    table = step.Properties["TableB"]
    for i in range(1, 10):
        table.AddRow()
        table.Properties["Integer"].Value = i
        table.Properties["FileOpen"].Value = "super"
        table.Properties["Number"].Value = 45.21
        table.Properties["ReadOnly"].Value = 777
        table.Properties["myselect"].Properties["TestStr"].Value = 777
        table.Properties["myselect"].Properties["Number B"].Value = 777
        table.SaveActiveRow()

def CreateGeometry(step):
    global geoSystem
    template1 = GetTemplate(TemplateName="Geometry")
    geoSystem = template1.CreateSystem()
    geometry1 = geoSystem.GetContainer(ComponentName="Geometry")
    geometry1.SetFile(FilePath=step.Properties["definition/filename"].Value)

def DeleteGeometry(step):
    global geoSystem
    geoSystem.Delete()

def RefreshMechanical(step):
    tree = step.UserInterface.GetComponent("Tree")
    root = tree.CreateTreeNode("Root")
    node1 = tree.CreateTreeNode("Node1")
    node2 = tree.CreateTreeNode("Node2")
    node3 = tree.CreateTreeNode("Node3")
    root.Values.Add(node1)
    root.Values.Add(node2)
    node2.Values.Add(node1)
    node2.Values.Add(node3)
    root.Values.Add(node3)
    tree.SetTreeRoot(root)
    chart = step.UserInterface.GetComponent("Chart")
    chart.Plot([1,2,3,4,5],[10,4,12,13,8],"b","Line1")
    chart.Plot([1,2,3,4,5],[5,12,7,8,11],"r","Line2")

def CreateMechanical(step):
    global dsSystem, geoSystem
    template2 = GetTemplate(
        TemplateName="Static Structural",
        Solver="ANSYS")
    geometryComponent1 = geoSystem.GetComponent(Name="Geometry")
    dsSystem = template2.CreateSystem(
        ComponentsToShare=[geometryComponent1],
        Position="Right",
        RelativeTo=geoSystem)
    if step.Properties["name"].Value=="error":
        raise UserErrorMessageException("Invalid system name. Please try again.")
    dsSystem.DisplayText = step.Properties["name"].Value

def DeleteMechanical(step):
    global dsSystem
    dsSystem.Delete()

def CreateFluent(step):
    global dsSystem, fluentSystem
    template3 = GetTemplate(TemplateName="Fluid Flow")
    geometryComponent2 = dsSystem.GetComponent(Name="Geometry")
    solutionComponent1 = dsSystem.GetComponent(Name="Solution")
    componentTemplate1 = GetComponentTemplate(Name="CFDPostTemplate")
    fluentSystem = template3.CreateSystem(
        ComponentsToShare=[geometryComponent2],
        DataTransferFrom=[Set(FromComponent=solutionComponent1, TransferName=None,
ToComponentTemplate=componentTemplate1)],
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
        Position="Right",
        RelativeTo=dsSystem)
    if step.Properties["name"].Value=="error":
        raise Exception("Invalid system name. Please try again.")
    fluentSystem.DisplayText = step.Properties["name"].Value

def CreateDialog(step):
    comp = step.UserInterface.Panel.CreateOKCancelDialog("MyDialog", "MyTitle", 400, 150)
    comp.SetOkButton("Ok")
    comp.SetMessage("My own message")
    comp.SetCallback(cbDialog)
    prop = step.Properties["nextstep"]
    prop.Options.Clear()
    s = step.NextStep
    val = s.Caption
    while s!=None:
        prop.Options.Add(s.Caption)
        s = s.NextStep
    prop.Value = val

def cbDialog(sender, args):
    dialog = CurrentWizard.CurrentStep.UserInterface.GetComponent("MyDialog").Hide()

def ValidateDialog(step, prop):
    dialog = step.UserInterface.GetComponent("MyDialog")
    dialog.Show()

def worker(step):
    progressDialog = step.UserInterface.GetComponent("Progress")
    progress = progressDialog.GetFirstOrDefaultComponent()
    progress.Reset()
    stopped = progress.UpdateProgress("Start progress...", 0, True)
    progressDialog.Show()
    for i in range(100):
        System.Threading.Thread.Sleep(100)
        stopped = progress.UpdateProgress("Start progress...", i+1, True)
        if stopped:
            break
    progressDialog.Hide()

def ValidateDialogProgress(step, prop):
    thread = System.Threading.Thread(System.Threading.ParameterizedThreadStart(worker))
    thread.Start(step)

def ValidateNextStep(step, prop):
    prop = step.Properties["nextstep"]
    s = step.NextStep
    v = False
    while s!=None:
        if prop.Value==s.Caption:
            v = True
        s.IsEnabled = v
        s = s.NextStep
    steps = step.UserInterface.GetComponent("Steps")
    steps.UpdateData()
    steps.Refresh()

def RefreshReport(step):
    report = step.UserInterface.GetComponent("Report")
    report.SetHtmlContent(System.IO.Path.Combine(ExtAPI.Extension.InstallDir,"help","report.html"))
    report.Refresh()

def EmptyReset(step):
    pass

def LogReport(step):
    ExtAPI.Log.WriteMessage("Report:")
    for s in step.Wizard.Steps.Values:
        ExtAPI.Log.WriteMessage("Step "+s.Caption)
        for prop in s.AllProperties:
            ExtAPI.Log.WriteMessage(prop.Caption+": "+prop.DisplayString)
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
import random
import math

def RefreshCharts(step):
    graph = step.UserInterface.GetComponent("Graph")
    graph.Title("Line Bar Graph")
    graph.ShowLegend(False)
    graph.Plot([-1, 0, 1, 2, 3, 4],  [0.5, -0.5, 0.5, -0.5, 0.5, 0.5], key="Variable A", color='g')
    graph.Bar([-1, 0, 1, 2, 3, 4],  [10, 20, 30, 10, 5, 20], key="Variable B")

    graphB = step.UserInterface.GetComponent("GraphB")
    graphB.Title("Plot Graph")
    graphB.YTickFormat("0.2f")

    xValues = []
    yValues = []
    for i in range(0, 100):
        xValues.append(i)
        yValues.append(abs(math.sin(i*0.2))*i/100.0)
    graphB.Plot(xValues, yValues, key="y = a*sin(bx)", color="c")
    graphB.Plot(xValues, yValues, key="y = x", color="m")

    xValues = []
    yValues = []
    for i in range(0, 100):
        xValues.append(i)
        yValues.append(i/100.0)
    graphB.Plot(xValues, yValues, key="y = x", color="m")

    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.2, 0.2, 0.3, 0.3], key="Smth", color="r")
    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.1, 0.3, 0.5, 0.7], key="Smth", color="r")
    graphB.Plot([0, 10, 20, 30, 100], [0.2, 0.2, 0.2, 0.3, 0.3])

    graphC = step.UserInterface.GetComponent("GraphC")
    graphC.Title("Pie Graph")
    graphC.Pie([1, 2, 3])
    graphC.Pie([20, 30, 5, 15, 12], [0, "Banana", 2, 3, "42"])

    graphD = step.UserInterface.GetComponent("GraphD")
    graphD.Title("Bar Graph")
    graphD.Bar(["Banana"], [70], key="key")
    graphD.Bar([0, "Banana", 2, 3, 4], [20, 30, 5, 15, 12], key="key")

    graphE = step.UserInterface.GetComponent("GraphE")
    graphE.Title("Bubble Graph")
    graphE.XTickFormat("f")
    graphE.YTickFormat("f")
    keys = ["one", "two", "three", "four", "five"]
    colors = ["#BB3333", "#33BB33", "#3333BB", "#BBBB33", "#BB33BB"]
    for c in range(0, 5):
        xValues = []
        yValues = []
        sizeValues = []
        for i in range(0, (c+1)*20):
            rad = random.randrange(c+1, c+2) + (random.random()*2-1)
            angle = random.random() * 2 * math.pi
            xValues.append(math.cos(angle) * rad)
            yValues.append(math.sin(angle) * rad)
            sizeValues.append(random.random() * 2.0 + 0.5)
        graphE.Bubble(xValues, yValues, sizeValues, key=keys[c], color=colors[c])

def CreateStaticStructural(step):
    template1 = GetTemplate(
        TemplateName="Static Structural",
        Solver="ANSYS")
    system1 = template1.CreateSystem()
    system1.DisplayText = "toto"

    nextStep = step.NextStep
    if nextStep!=None:
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Model"
    nextStep = nextStep.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Model"

def OnSelectContext(step, prop):
    firstDMStep = step.NextStep
    secondDMStep = firstDMStep.NextStep
    firstSCStep = secondDMStep.NextStep
    secondSCStep = firstSCStep.NextStep

    firstGeoStep = step.NextStep
    if prop.Value == "DesignModeler":
        firstDMStep.IsEnabled = True
        secondDMStep.IsEnabled = True
        firstSCStep.IsEnabled = False
        secondSCStep.IsEnabled = False
    elif prop.Value == "SpaceClaim":
        firstDMStep.IsEnabled = False
        secondDMStep.IsEnabled = False
        firstSCStep.IsEnabled = True
        secondSCStep.IsEnabled = True

    panel = step.UserInterface.Panel.GetComponent("Steps")
    panel.UpdateData()
    panel.Refresh()


def RefreshResultsProject(step):
    step.Properties["Res"].Value = ExtAPI.Extension.Attributes["result"]
    panel = step.UserInterface.Panel.GetComponent("Properties")
    panel.UpdateData()
    panel.Refresh()
```

For a mixed wizard, the definition of the first step executed in the **Project** tab specifies the Ansys products from which subsequent steps are executed. For instance, in the following excerpted code, multiple actions are defined. Steps in the mixed wizard call these actions.

```
…
def action1(step):

    template1 = GetTemplate( TemplateName="Static Structural", Solver="ANSYS")
    system1 = template1.CreateSystem()
    geometry1 = system1.GetContainer(ComponentName="Geometry")
    geometry1.SetFile(FilePath=step.Properties["filename"].Value)

    nextStep = step.NextStep
    if nextStep!=None:
        nextStep.SystemName = system1.Name
        nextStep.ComponentName = "Geometry"
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
    thirdStep = step.Wizard.Steps["Step3"]
    if thirdStep!=None:
        thirdStep.SystemName = system1.Name
        thirdStep.ComponentName = "Model"
…
```

Mixed Wizard Example

In addition to the project wizard described in the previous section, the extension **`WizardDemos`** includes multiple target product wizards and a mixed wizard named **`BridgeSimulation`**.

The ACT customization guides for DesignModeler, SpaceClaim, and Mechanical describe the target product wizards. The concluding section (p. 183) provides links to these guides. The following topics describe the mixed wizard **`BridgeSimulation`**:

Defining the Mixed Wizard

The mixed wizard **`BridgeSimulation`** executes one step in the Workbench **Project** tab, runs the wizard **`CreateBridge`** in either DesignModeler or SpaceClaim, runs the wizard **`SimpleAnalysis`** in Mechanical, and then returns to the Project tab to execute the step **`Results`**.

An excerpt from the file `WizardDemos.xml` follows. Code is omitted for the element **`<uidefinition>`** and all wizards other than the mixed wizard **`BridgeSimulation`**.

```
<extension version="2" minorversion="1" name="WizardDemos">
 <guid shortid="WizardDemos">7fdb141e-3383-433a-a5af-32cb19971771</guid>
 <author>Ansys, Inc.</author>
 <description>Simple extension to test wizards in different contexts.</description>

 <script src="main.py" />
 <script src="ds.py" />
 <script src="dm.py" />
 <script src="sc.py" />

 <interface context="Project|Mechanical|SpaceClaim">
  <images>images</images>
 </interface>

 <interface context="DesignModeler">
  <images>images</images>

  <toolbar name="Deck" caption="Deck">
   <entry name="Deck" icon="deck">
    <callbacks>
     <onclick>CreateDeck</onclick>
    </callbacks>
   </entry>
   <entry name="Support" icon="Support">
    <callbacks>
     <onclick>CreateSupport</onclick>
    </callbacks>
   </entry>
  </toolbar>

 </interface>

 <simdata context="DesignModeler">
  <geometry name="Deck" caption="Deck" icon="deck" version="1">
```

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
   <callbacks>
    <ongenerate>GenerateDeck</ongenerate>
   </callbacks>
   <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
   <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
   <property name="Beams" caption="Beams" control="integer" default="31" />
  </geometry>
 </simdata>

 <simdata context="DesignModeler">
  <geometry name="Support" caption="Support" icon="support" version="1">
   <callbacks>
    <ongenerate>GenerateSupport</ongenerate>
   </callbacks>
   <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
   <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
   <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
   <property name="Number" caption="Number" control="integer" default="3" />
  </geometry>
 </simdata>

 ...

 <wizard name="BridgeSimulation" version="1" context="Project" icon="bridge">
  <description>Simple wizard for mixed wizard demonstration.</description>

  <step name="Project" caption="Create Project" version="1" context="Project" HelpFile="help/prj1.html">
   <description>Create a static structural analysis.</description>

   <callbacks>
    <onupdate>CreateStaticStructural</onupdate>
    <!--<onreset>DeleteStaticStructural</onreset>-->
   </callbacks>

   <property name="Name" caption="system name" control="text" />
   <property name="Context" caption="geometry context" control="select">
    <attributes options="DesignModeler,SpaceClaim"/>
    <callbacks>
     <onvalidate>OnSelectContext</onvalidate>
    </callbacks>
   </property>

  </step>

  <step name="DeckDM" caption="DeckDM" version="1" context="DesignModeler" HelpFile="help/dm1.html">
   <description>Create the deck.</description>

   <callbacks>
    <onupdate>UpdateDeck</onupdate>
   </callbacks>

   <propertygroup display="caption" name="Deck" caption="Deck Definition" >
    <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
    <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
    <property name="Beams" caption="Beams" control="integer" default="31" />
   </propertygroup>

  </step>

  <step name="SupportsDM" caption="SupportsDM" context="DesignModeler" enabled="true" version="1"
HelpFile="help/dm2.html">
   <description>Create supports.</description>

   <callbacks>
    <onupdate>UpdateSupports</onupdate>
   </callbacks>

   <propertygroup display="caption" name="Supports" caption="Supports Definition" >
    <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
    <property name="Number" caption="Number" control="integer" default="3" />
   </propertygroup>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.

```
</step>

<step name="DeckSC" caption="DeckSC" version="1" context="SpaceClaim">
 <description>Create the deck.</description>

 <callbacks>
  <onupdate>UpdateDeckSC</onupdate>
 </callbacks>

 <propertygroup display="caption" name="Deck" caption="Deck Definition" >
  <property name="Length" caption="Length" control="float" unit="Length" default="300 [m]" />
  <property name="Width" caption="Width" control="float" unit="Length" default="20 [m]" />
  <property name="Beams" caption="Beams" control="integer" default="31" />
 </propertygroup>
</step>

<step name="SupportsSC" caption="SupportsSC" context="SpaceClaim" enabled="true" version="1">
 <description>Create supports.</description>

 <callbacks>
  <onupdate>UpdateSupportsSC</onupdate>
 </callbacks>

 <propertygroup display="caption" name="Supports" caption="Supports Definition" >
  <property name="Height" caption="Height" control="float" unit="Length" default="100 [m]" />
  <property name="Number" caption="Number" control="integer" default="3" />
 </propertygroup>
</step>


<step name="Mesh" caption="Mesh" version="1" context="Mechanical" HelpFile="help/ds1.html">
 <description>Setup some mesh controls.</description>

 <callbacks>
  <onreset>RemoveControls</onreset>
  <onupdate>CreateMeshControls</onupdate>
 </callbacks>

 <propertygroup display="caption" name="Sizing" caption="Mesh Sizing" >
  <property name="Location" caption="Edge Location" control="geometry_selection">
   <attributes selection_filter="edge" />
   <callbacks>
    <isvalid>IsLocationValid</isvalid>
   </callbacks>
  </property>
  <property name="Ndiv" caption="Divisions" control="integer" />
 </propertygroup>

</step>

<step name="Solution" caption="Solution" version="1" context="Mechanical" HelpFile="help/ds2.html">
 <description>Setup loads.</description>

 <callbacks>
  <onrefresh>RefreshLoads</onrefresh>
  <onreset>RemoveLoads</onreset>
  <onupdate>CreateLoads</onupdate>
 </callbacks>

 <propertygroup display="caption" name="Mesh" caption="Mesh Statistics" >
  <property name="Nodes" caption="Nodes" control="text" readonly="true" />
  <property name="Elements" caption="Elements" control="text" readonly="true" />
 </propertygroup>
 <propertygroup display="caption" name="FixedSupport" caption="Fixed Support" >
  <property name="Location" caption="Face Location" control="geometry_selection">
   <attributes selection_filter="face" />
   <callbacks>
    <isvalid>IsLocationFSValid</isvalid>
   </callbacks>
  </property>
```

```
   </propertygroup>

  </step>

  <step name="Results" caption="Results" version="1" context="Project" HelpFile="help/prj2.html">
   <description>View Results.</description>

   <callbacks>
    <onrefresh>RefreshResultsProject</onrefresh>
   </callbacks>

   <property name="Res" caption="Deformation" control="text" readonly="true" />

  </step>

 </wizard>

</extension>
```

Understanding the elements **<interface>** and **<simdata>** is necessary to understanding the mixed wizard **BridgeSimulation**.

Wizard Interface Definition

The element **<interface>** defines two user interfaces for the extension **WizardDemos**, both of which the wizard **BridgeSimulation** uses.

- The first element **<interface>** has the attribute **context** set to **Project|Mechanical|SpaceClaim**, which indicates that wizards with their contexts set to any of these products use this interface.
- The second element **<interface>** has the attribute **context** set to **DesignModeler**, which indicates that wizards with their contexts set to this product use this interface. This element **<interface>** has a child element **<toolbar>** that defines two toolbar buttons for exposure in DesignModeler. When the buttons are clicked, the callback **<onclick>** executes the functions **CreateDeck** and **CreateSupport**, creating a deck geometry with supports.

Simdata Definition

The element **<simdata>** provides data. This extension has two such elements to provide data for creating the geometries **Deck** and **Support** in DesignModeler.

Wizard Definition

The element **<wizard>** named **BridgeSimulation** defines the mixed wizard **BridgeSimulation**. The attribute **context** specifies the product in which the wizard is executed. Because this wizard accesses the target product from the **Project** tab instead of executing in the target product directly, **context** is set to **Project**.

Step Definition

The element **<step>** defines a step in the mixed wizard **BridgeSimulation**. This wizard has eight steps: **Project**, **DeckDM**, **SupportsDM**, **DeckSC**, **SupportsSC**, **Mesh**, **Solution**, and **Results**. The steps with **DM** in their names are executed in DesignModeler. The steps with **SC** in their names are executed in SpaceClaim.

For each step:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- The attributes **name**, **version**, and **caption** specify the name, version, and display text for the step.
- The attribute **context** specifies the Ansys product in which the specific step is executed. You only need to set the context for the step if it differs from the context for the wizard.
- The attribute **HelpFile** specifies the HTML file to display as the custom help for the step.
- The element **<callbacks>** specifies callbacks to functions defined in all four IronPython scripts referenced in this extension's XML file.
  - For the step **Project**, the callback **<onupdate>** executes the function **CreateStaticStructural** in the script `main.py`.
  - For the step **DeckDM**, the callback **<onupdate>** executes the function **UpdateDeck** in the script `dm.py`.
  - For the step **SupportsDM**, the callback **<onupdate>** executes the function **UpdateSupports** in the script `dm.py`.
  - For the step **DeckSC**, the callback **<onupdate>** executes the function **UpdateDeckSC** in the script `sc.py`.
  - For the step **SupportsSC**, the callback **<onupdate>** executes the function **UpdateSupportsSC** in the script `sc.py`.
  - For the step **Mesh**, the callbacks **<onreset>** and **<onupdate>** execute the functions **RemoveControls** and **CreateMeshControls** in the script `ds.py`.
  - For the step **Solution**, the callback **<onrefresh>**, **<onreset>**, and **<onupdate>** execute the functions **RefreshLoads**, **RemoveLoads**, and **CreateLoads** in the script `ds.py`.
  - For the step **Results**, the callback **<onrefresh>** executes the function **RefreshResultsProject** in the script `main.py`.

Defining Functions for the Mixed Wizard

This mixed wizard incorporates steps from the Workbench **Project** tab with steps from target product wizards for DesignModeler, Mechanical, and SpaceClaim. Consequently, the callbacks for steps execute functions across all four IronPython scripts referenced by this extension's XML file:

- `main.py`
- `dm.py`
- `ds.py`
- `sc.py`

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Custom Wizard Help Files

To provide custom help for a wizard, you create HTML files that you then store in a child folder in the extension's folder. In the XML file, the element **<step>** has the attribute **helpFile**. You use this attribute to reference the relative path for the HTML file to display in the step's help panel.

Custom help files can contain any valid HTML5 syntax and supported media, such as images and graphs. You simply place the supported media files in the same folder as the HTML files.

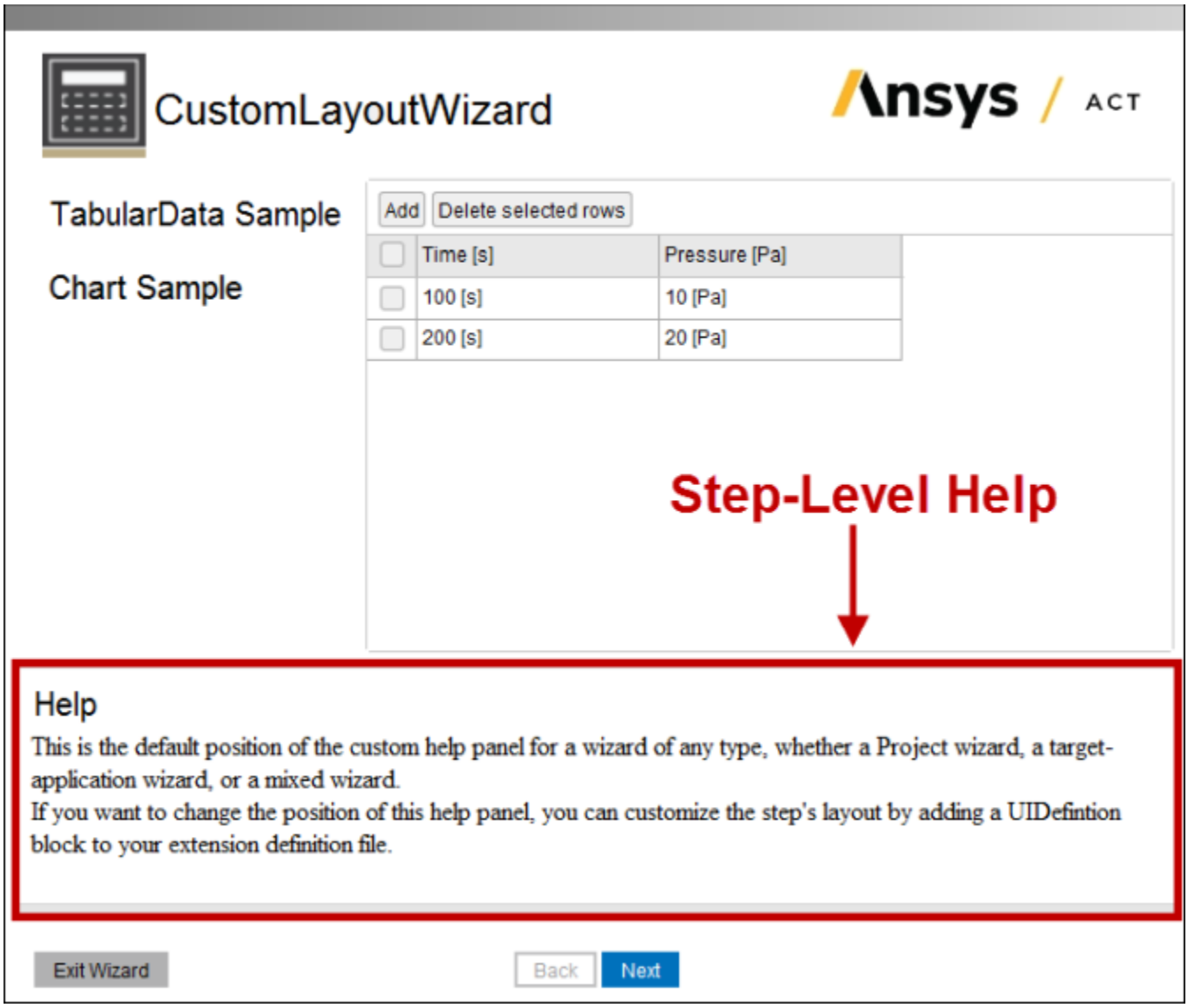
In the following example, the help file **step1.html** is located in the folder **wizardhelp** within the extension folder.

```
<step name="Geometry" caption="Geometry" version="1" helpFile="wizardhelp/step1.html">
```

If you give the help files for steps the same names as the steps themselves and then store these files in a folder named **help**, you do not need to use the attribute **helpFile** to reference them. The extension automatically finds and displays the appropriate help file for each step.

The custom help for a Workbench-based wizard displays by default in a help panel at the bottom of the wizard. However, you can customize the wizard interface (p. 162). For example, you could change the location of this panel.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



To provide custom help for the properties that are defined within a step, you can either create HTML files or define help text directly in the extension's XML file:

Defining Help for Properties in HTML Files

Defining Help for Properties Directly in the XML File

Defining Help for Properties in HTML Files

If you choose to create HTML files for properties, you must give them the same name as the step and the property, separated by an underscore. For example, assume that **`MeshingStep`** is the step name and **`MeshResolution`** is the property name. The help file for the step must be named **`MeshingStep`**, and the help file for the property must be named **`MeshingStep_MeshResolution`**. When you use these file-naming conventions, you do not need to use the attribute **`helpFile`** to explicitly

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

reference the HTML file for a step. The extension can automatically find and display the custom help for steps and their properties.

Note:

For HTML files for properties, the recommended practice is to avoid images and to limit the text to no more than 200 characters.

Defining Help for Properties Directly in the XML File

If you choose to define help text for properties directly in the extension's XML file, you use the element **<help>** in the property definitions. For example, assume that a step has a property named **geometryfile**. You might define custom help for this property in the element **<help>** as follows:

```
<property name="geometryfile" caption="Geometry file" control="fileopen" default="">
    <help>Select a geometry source file to import. Note the geometry should contain the flow volume in addition
      <callbacks>
          <isvisible>isVisibleGeometryFile</isvisible>
          <isvalid>isValidGeometryFile</isvalid>
      </callbacks>
 </property>
```

Custom Wizard Interfaces

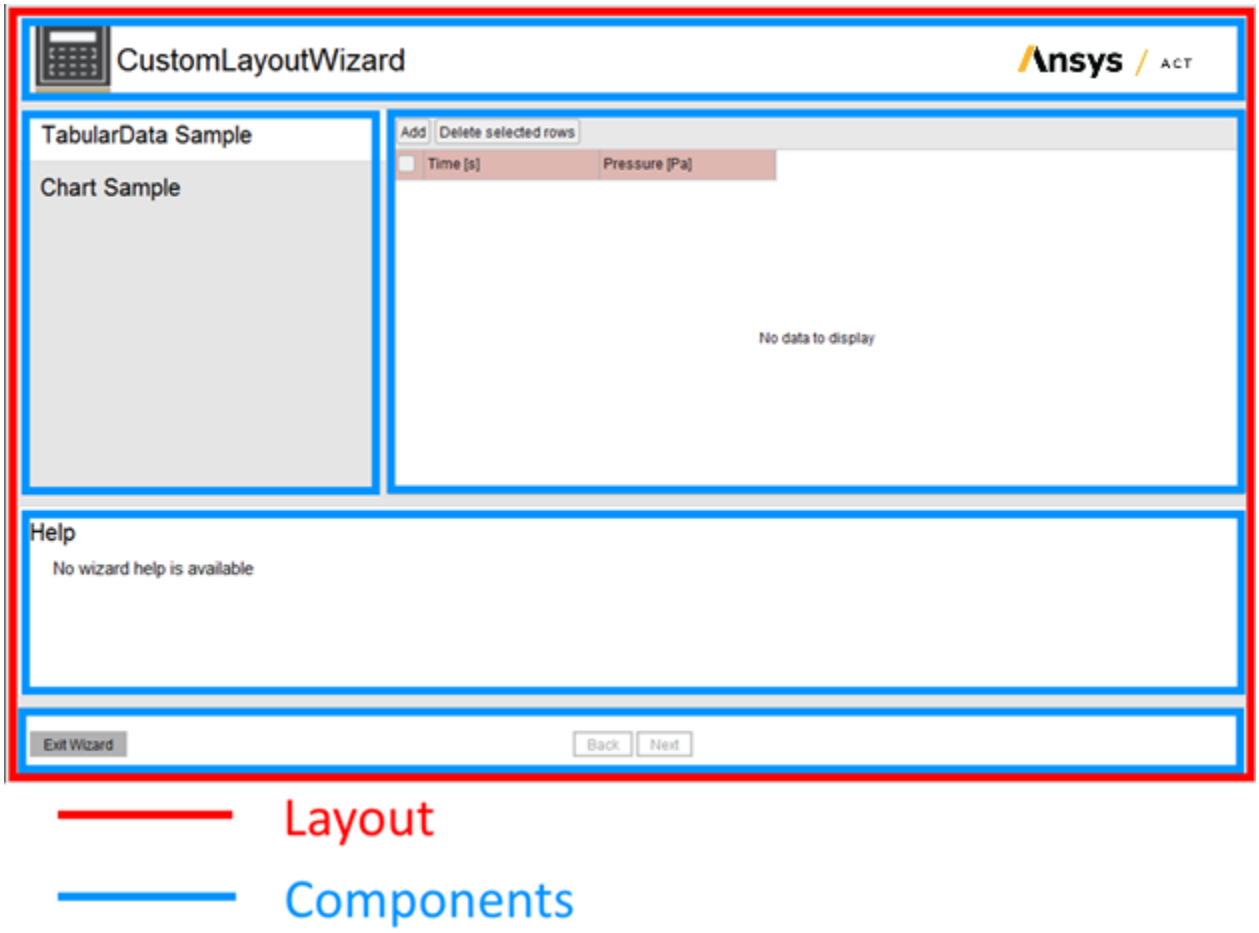
ACT provides you with the capability to customize the interface of a Workbench-based wizard, defining layouts both for the interface as a whole and for the individual interface components within the layout.

In the extension's XML file, you use the optional element **<uidefinition>** to define wizard interface customizations. The basic definition of this element follows:

```
<uidefinition>
     <layout>
        <component/>
       <component/>
       <component/>
       <component/>
     </layout>
</uidefinition>
```

The customizations are exposed as follows:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



User Interface Definition

The element **<uidefinition>** defines a single user interface for the wizard. It is the top-level entry that contains all layout definitions.

Layout Definition

The element **<layout>** defines a single custom layout for the wizard. A separate layout can be defined for each step in the wizard. The attribute **name** defines the name of the layout. The name is referenced in a wizard step in the following format:

```
LayoutName@ExtensionName
```

This notation allows a wizard to reference a layout defined in a different extension.

Component Definition

The element **<component>** defines a single component in a custom layout for the wizard.

- The mandatory attribute **name** defines the name for the component. The component name is required to position it next to another component. To do this, the name is used in conjunction with the following attributes:
  - **leftAttachment**

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

  - **rightAttachment**
  - **bottomAttachment**
  - **topAttachment**

  The component name is also used in the IronPython script with the method **GetComponent()**. For instance, in a step, the callback **Onrefresh** can use the following code:

```
component = step.UserInterface.GetComponent(ComponentName)
```

- The mandatory attribtues **heightType**, **height**, **widthType**, and **width** define the dimensions of the component. Possible values are as follows:
  - **FitToContent**: The component is to have a default size that is set by ACT. When selected, the attributes **height** and **width** become irrelevant.
  - **Percentage**: The height and width of the component is expressed as percentages.
  - **Fixed**: The height and width of the component is expressed in pixels.
- The mandatory attribute **componentType** defines the type of component. Some examples of possible component types are:
  - **startPageHeaderComponent** (Defines banner titles)
  - **propertiesComponent**
  - **chartComponent**
  - **tabularDataComponent**
  - **buttonsComponent**
  - **stepsListComponent**
- The attributes **leftOffset**, **rightOffset**, **bottomOffset**, and **topOffset** specify the distance and the margin between the different components.

The next section provides an example of customizing a wizard interface.

Custom Wizard Interface Example

You use the optional element **<uidefinition>** to create layouts for your custom wizard interface. To demonstrate, the supplied extension **CustomLayout** includes a project wizard named **CustomLayoutWizard**. This extension defines two custom layouts—one for each step in the **CustomLayoutWizard** wizard.

Because the basics of constructing a wizard are described elsewhere, the following topics focus solely on aspects of layout customization:

Defining the Custom Wizard Interface

Defining Functions for the Custom Wizard Layout

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Reviewing the Custom Wizard Layouts

Defining the Custom Wizard Interface

The file **CustomLayout.xml** follows.

```
<extension name="CustomLayout" version="0">
  <guid shortid="CustomWizardLayout">e7fa4157-90b7-4156-ac31-4000d1d687e1</guid>
  <script src="main.py" />
  <wizard name="CustomLayoutWizard" caption="CustomLayoutWizard" version="1" context="Project">
    <step layout="TabularDataLayout@CustomLayout" name="TabularData Sample" version="0" caption="TabularData Sam
      <propertytable display="Worksheet" control="tabulardata" name="Table" caption="Table" persistent="False" p
        <property control="quantity" name="Time" caption="Time" persistent="False" parameterizable="False" unit=
        <property control="quantity" name="Pressure" caption="Pressure" persistent="False" parameterizable="Fals
      </propertytable>
      <callbacks>
        <onrefresh>onrefreshTabularDataSample</onrefresh>
        <onreset>onresetTabularDataSample</onreset>
      </callbacks>
    </step>
    <step layout="ChartLayout@CustomLayout" name="Chart Sample" version="0" caption="Chart Sample">
      <callbacks>
        <onrefresh>onrefreshChartSample</onrefresh>
      </callbacks>
    </step>
  </wizard>
  <uidefinition>
    <layout name="TabularDataLayout">
      <component name="Title" leftoffset="10" topoffset="10" rightoffset="10" bottomattachment="Steps" bottomoff
      <component name="Steps" leftoffset="10" topattachment="Title" topoffset="0" rightattachment="TabularData"
      <component name="TabularData" leftattachment="Steps" leftoffset="0" topattachment="Title" topoffset="0" ri
      <component name="Help" leftoffset="10" topattachment="TabularData" topoffset="0" rightoffset="10" bottomat
      <component name="Submit" leftoffset="10" topattachment="Help" topoffset="0" rightoffset="10" bottomoffset=
    </layout>
    <layout name="ChartLayout">
      <component name="Title" leftoffset="10" topoffset="10" rightoffset="10" bottomattachment="Steps" bottomoff
      <component name="Steps" leftoffset="10" topattachment="Title" topoffset="0" rightattachment="Chart" righto
      <component name="Chart" leftattachment="Steps" leftoffset="0" topattachment="Title" topoffset="0" rightoff
      <component name="Submit" leftoffset="10" topattachment="Chart" topoffset="0" rightoffset="10" bottomoffset
    </layout>
  </uidefinition>
  <description>Demo layout</description>
</extension>
```

In this file, the element **<uidefinition>**, defines two layouts:

- **TabularDataLayout**
  - This layout has five components: **Title**, **Steps**, **TabularData**, **Help**, and **Submit**.
  - This layout is referenced by the step **TabularData Sample** with **TabularDataLayout@CustomLayout**.
- **ChartLayout**
  - This layout has four components: **Title**, **Steps**, **Chart**, and **Submit**.
  - This layout is referenced by the step **Chart Sample** with **ChartLayout@CustomLayout**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Defining Functions for the Custom Wizard Layout

The script `main.py` follows. It defines all functions executed by the callbacks in the project wizard **`CustomLayoutWizard`**. These functions reference the defined layouts using **`GetComponent()`**.

```
def onrefreshTabularDataSample(step):
    comp = step.UserInterface.GetComponent("TabularData")
    table = step.Properties["Table"]
    comp.SetPropertyTable(table)

def onrefreshChartSample(step):
    table = step.PreviousStep.Properties["Table"]
    tableValue = table.Value
    rowCount = table.RowCount

    x = []
    y = []
    for rowIndex in range(0, rowCount):
        x.append(tableValue["Table/Time"][rowIndex].Value.Value)
        y.append(tableValue["Table/Pressure"][rowIndex].Value.Value)

    comp = step.UserInterface.GetComponent("Chart")
    comp.Plot(x, y)

def onresetTabularDataSample(step):
    #nothing to do
    pass
```

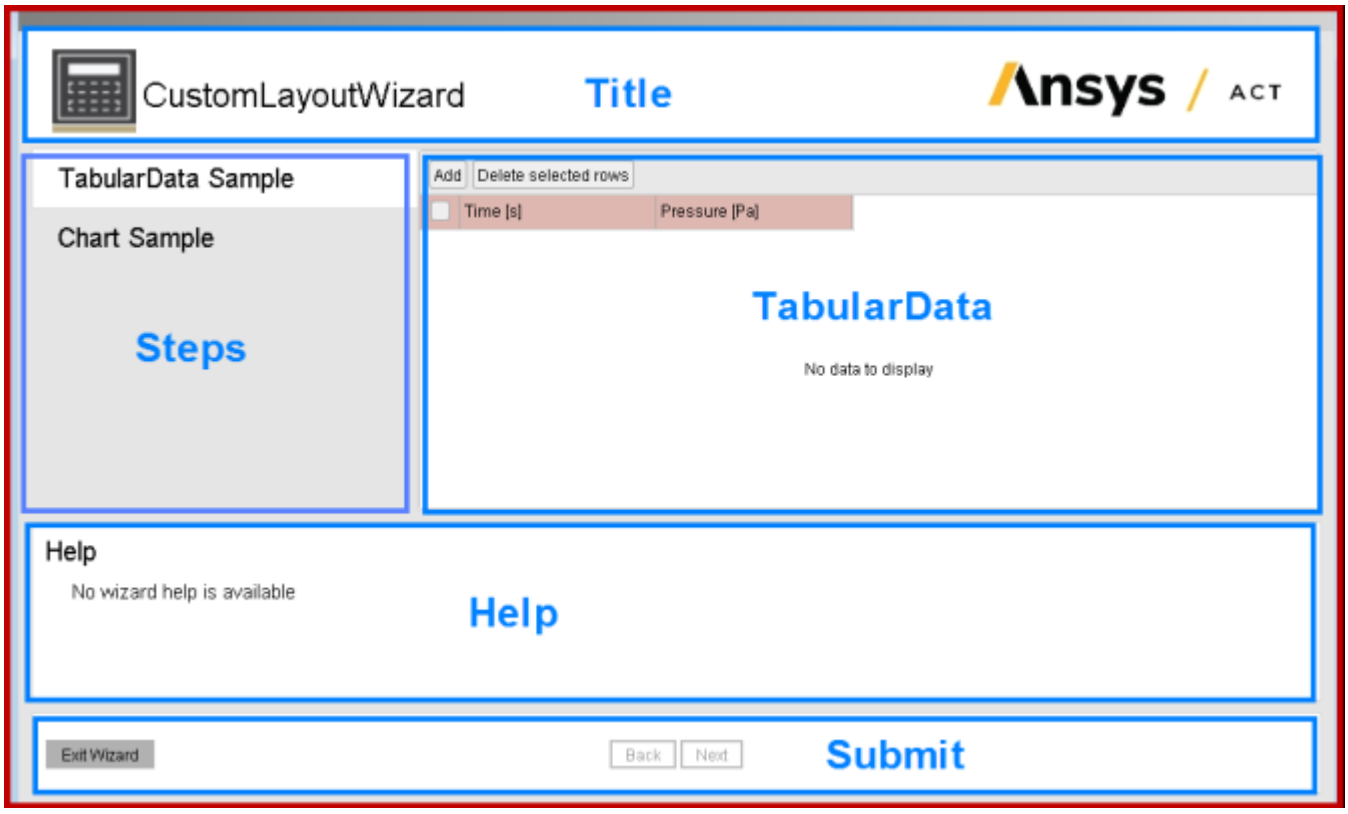
Reviewing the Custom Wizard Layouts

The following images show the exposure of the custom wizard layouts defined in the project wizard **`CustomLayoutWizard`**. In both images, the wizard layout is noted in red and individual components are noted in blue.

The layout **`TabularDataLayout`** is used for the step **`TabularData Sample`**.

Data must be entered into the component **`TabularData`** for the **`Next`** button in the component **`Submit`** to become enabled.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



The layout **`ChartLayout`** is used for the step **`Chart Sample`**.

The component **`Chart`** contains the chart generated for the tabular data entered in the previous step.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.

Debugging

This section provides information about debugging IronPython scripts during extension development:

Debug Mode

ACT Debugger

Debugging with Microsoft® Visual Studio

A short video shows how to load the supplied extension (p. 81) **`DebuggerDemo`** in Workbench, add a custom **Pre-Design** system and a **Static Structural** analysis system in the **Project Schematic**, and then view the properties of the **Pre-Geometry** cell, which determine how the model is created. The video then explains how to start the ACT Debugger (p. 170) and use some of its features. If you are new to debugging scripts, you can follow along with this video by performing the same actions to learn how to use the **ACT Debugger**. To watch this video now, click here.

Note:

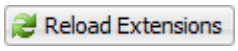
The **ACT Debugger** currently supports debugging scripts for extensions that execute from the **Project** page in Workbench and from DesignModeler and Mechanical. It does not yet support debugging scripts on the Linux platform.

Debug Mode

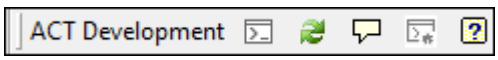
Regardless of whether you intend to debug scripts in the ACT Debugger (p. 170) or in Microsoft Visual Studio (p. 181), you must first enable debug mode. To accomplish this, from the Workbench menu, you select **Tools** → **Options** → **Extensions** and then select the **Debug Mode** check box. You can also select this check box in the settings that become available when you click the gear icon in the graphic-based Extension Manager (p. 41), which is started from the ACT Start Page (p. 39).

Once debug mode is enabled, you can debug extensions. To help you debug, the debug mode exposes the following features:

- In Workbench, the **Reload Extensions** button displays so that you can easily reload all loaded extensions.

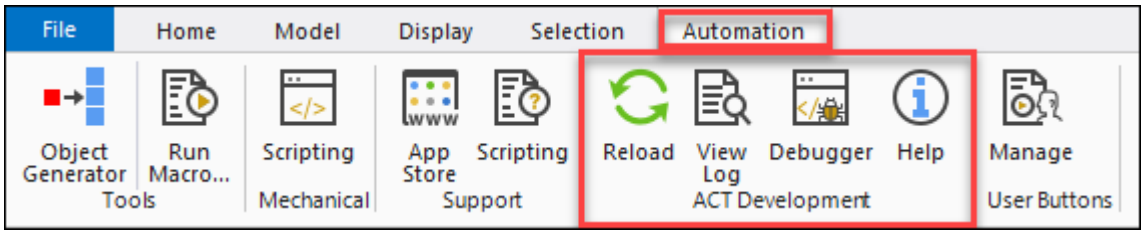


- In DesignModeler, the **ACT Development** toolbar displays. Clicking the second button reloads all loaded extensions. The other buttons open the **ACT Console**, the **Extensions Log File**, the **ACT Debugger**, and help panel provided on the **ACT Start Page**.



Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

- In Mechanical, the **ACT Development** group displays on the ribbon's **Automation** tab. This first button in this group reloads all loaded extensions. The other buttons open the **Extensions Log File**, **ACT Debugger**, and help panel provided on the **ACT Start Page**. To the left, the **ACT** group includes a button for opening the **ACT Console**. To the right of the **Automation** tab are tabs for any loaded extensions.



Access points for debugging are provided in DesignModeler, Mechanical, and the Workbench **Project** tab.

- In the Workbench **Project** tab, debug mode is enabled as soon as you select the **Debug Mode** check box. Currently, you cannot debug scripts on the Linux platform.

- In DesignModeler and Mechanical, debug mode is not enabled until you restart the product. After selecting the **Debug Mode** check box, you must exit and restart the product to make debugging functionality available.

When debug mode is enabled, you can use a debugger to observe the run-time behavior of IronPython scripts in loaded extensions. You typically use a debugger to either verify that correct values are being stored for variables or determine why an exception is thrown. Once you have finished debugging a scripted extension, you can use the binary extension builder (p. 49) to compile it into a binary extension (WBEX file) that you can share with others.

ACT Debugger

The **ACT Debugger** provides state-of-the-art capabilities for debugging the IronPython scripts for ACT extensions. With a graphical user interface that includes a simplified version of the ACT Console (p. 53), you can reproduce, diagnose, and resolve script issues faster and easier.

The following topics provide general usage information:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Starting the ACT Debugger

Before starting the **ACT Debugger**, first use the Extension Manager (p. 41) to install and load the extensions that you want to debug. Also ensure that debug mode (p. 169) is enabled. Otherwise, debugging options won't be available to you.

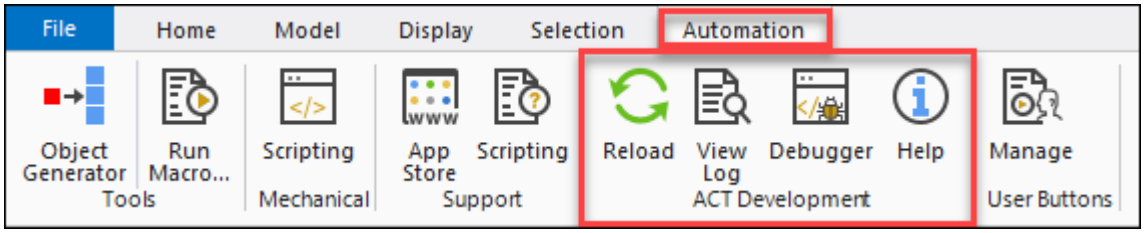
Because the **ACT Debugger** performs product-specific initialization, always start it from the Ansys product where you intend to execute the extension. This ensures that the debugger attaches to the correct process. For example, assume that you start the debugger from the Workbench **Project** tab and then try to debug an extension that is meant to be executed in Mechanical. Because you did not open the debugger from Mechanical, you cannot debug the Mechanical extension.

When the extension is to execute from the Workbench **Project** tab, start the debugger using one of the following methods:

- From the Workbench menu, select **Extensions** → **Open ACT Debugger**.
- From the ACT Start Page (p. 39), click **ACT Debugger**.

When the extension is to execute from DesignModeler, in the **ACT Development** toolbar, click the button for starting the debugger. In the following figure, this is the third button.

When the extension is to execute from Mechanical, in the **ACT Development** group on the ribbon's **Automation** tab, click the button for starting the debugger. In the following figure, this is the third button.



Once the debugger starts, the **Sources** pane displays a folder for each loaded extension, which contains its XML file and IronPython scripts. If you expand a folder and click a file, you can see and edit this file in the pane to the right.

To begin debugging a loaded extension, you expand the folder for the extension, click a script, and then set breakpoints (p. 173). When you click the button in the debugger toolbar for attaching to the Ansys product where the extension is to execute, the debugger begins listening to this product as it executes the underlying scripts. By viewing this product next to the debugger, you can observe the run-time behavior of the attached extension to reproduce unexpected behaviors, investigate issues, and test script changes.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

The debugger allows you to execute the extension while inspecting the logic and memory of the script, step by step, directly in the source file. You can pause execution at breakpoints, explore and edit variables, browse the stack of function calls, and even change the script.

Understanding Debugger Terms

This section provides an alphabetic listing of debugging terms.

- Breakpoints: Locations in the script where the debugger is to pause so that you can investigate the state of the extension.
- Call stack: List of executed method calls that have led to the current statement, ordered from deepest (most nested) to shallowest (top level).
- Console: Tool for interactively testing commands during development and debugging.
- Exceptions: Error states that occur while a script is executed. You want to pause the debugger when an exception is thrown so that you have a chance to examine it before a handler is invoked.
- Locals: Variables that are currently available for the active location in the script.
- Watches: Variables and expressions that you want to observe and evaluate during debugging.

Understanding the Debugger Interface

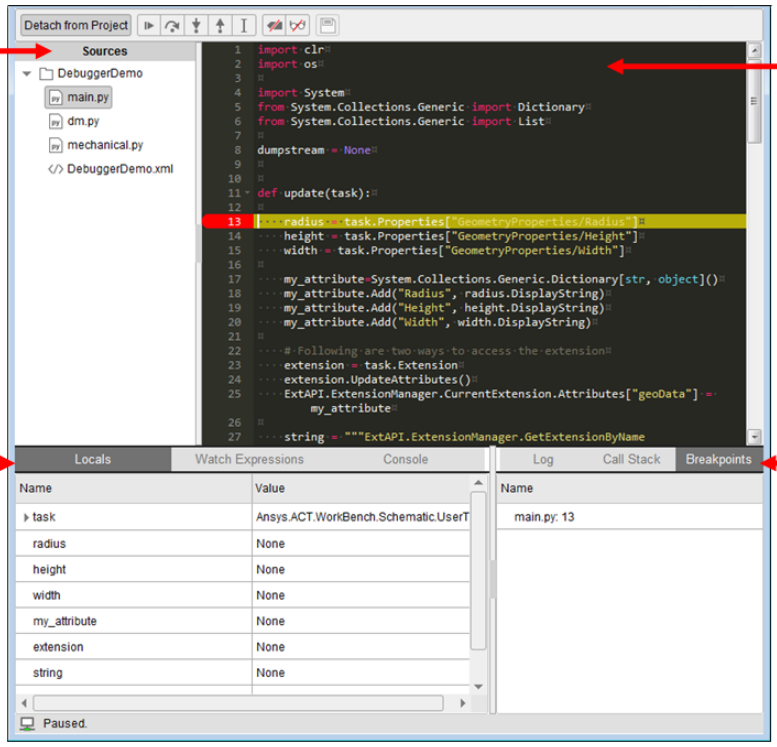
The **ACT Debugger** has four panes, two of which have multiple tabs. In the upper left corner, the **Sources** pane displays a folder for each extension that is loaded. If you load or unload extensions, the debugger refreshes this pane.

To view the files for a loaded extension, you expand the extension's folder. When you select a script, it opens in an editor in the pane to the right, where you can set breakpoints (p. 173).

If you edit or save an XML file or script, the debugger automatically reloads the extension, thereby providing an efficient method for interactive testing of your changes.

The following figure shows the supplied extension (p. 81) **`DebuggerDemo`** loaded. If you want, you can use the Extension Manager (p. 41) to install and load this extension. You can then follow along with the short video on the **ACT Debugger** to learn how to use some of its features.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.



In this figure, the debugger is started from the Workbench **Project** tab, and it is attached and listening to the extension **`DebuggerDemo`**. A breakpoint is set at line 13 in the script **`main.py`**, where the debugger is currently paused. Line numbers for lines with breakpoints are highlighted in red. The line at which the debugger is paused is highlighted in yellow.

Note:

The **Log** tab in the lower right pane displays the Extensions Log File (p. 47).

Setting Breakpoints

Breakpoints are essential to debugging. You set a breakpoint in each script location where you want to pause the debugger to investigate the current state of the extension. When the debugger encounters a breakpoint, it pauses extension execution, placing you in break mode. Most debugger features are available only in break mode.

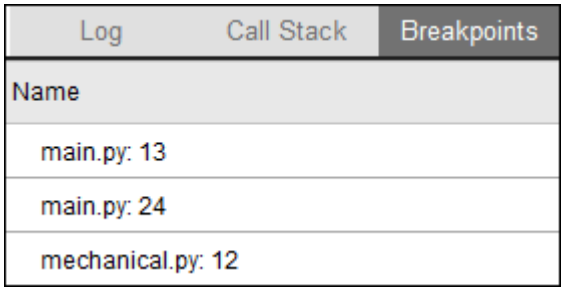
Repeatedly clicking to the left of a line number switches between setting and clearing a breakpoint. To set or clear breakpoints, do not click the line number but to the left of it. You can set a breakpoint on any line with an executable statement. While it is possible to set a breakpoint on a commented line, blank line, or a declaration of a namespace, class, or method, such a breakpoint serves no purpose.

In the extension **`DebuggerDemo`**, assume that you have inserted breakpoints at lines 13 and 24 in the script **`main.py`**.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

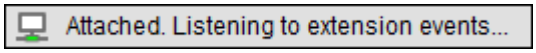
```
1  import clr
2  import os
3
4  import System
5  from System.Collections.Generic import Dictionary
6  from System.Collections.Generic import List
7
8  dumpstream = None
9
10
11 def update(task):
12
13     radius = task.Properties["GeometryProperties/Radius"]
14     height = task.Properties["GeometryProperties/Height"]
15     width = task.Properties["GeometryProperties/Width"]
16
17     my_attribute=System.Collections.Generic.Dictionary[str, object]()
18     my_attribute.Add("Radius", radius.DisplayString)
19     my_attribute.Add("Height", height.DisplayString)
20     my_attribute.Add("Width", width.DisplayString)
21
22     #Following are two ways to access the extension
23     extension = task.Extension
24     extension.UpdateAttributes()
25     ExtAPI.ExtensionManager.CurrentExtension.Attributes["geoData"] =
           my_attribute
26
27     string = """ExtAPI.ExtensionManager.GetExtensionByName("Debugger"
           ).GetModule().CreatePressureVesselDM("")"""
```

Also assume that you have set a breakpoint at line 12 in the script **`mechanical.py`**. In the lower right pane of the debugger, the **Breakpoints** tab displays all three breakpoints that are currently set, displaying the filename first and the line number second:



You can double-click any entry in the **Breakpoints** tab to have the debugger move to this script and line. This tab is especially handy when you are debugging large or complex scripts.

After setting one or more initial breakpoints, you click the button in the debugger toolbar for attaching to the product so that the debugger begins listening to the execution of the extension. In the lower left corner of the debugger, the status bar displays operational information.



At a breakpoint, you can use buttons in the debugger toolbar (p. 175) to progress through script statements to investigate the state of the extension. In break mode, you can also set new breakpoints

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

and clear existing ones. After you finish debugging, you can click the debugger toolbar button for clearing all breakpoints from the loaded scripts.

Using the Debugger Toolbar

Descriptions follow for all buttons in the debugger toolbar. You use these buttons to manage the debugging process. Regardless of navigation mode, the debugger always pauses at the next breakpoint.

Button	Shortcut Key	Action
Attach to Project or Detach from Project	---	Switches between attaching the debugger to and detaching it from an Ansys product. In addition to indicating whether you can attach or detach, the button label indicates the product from which the debugger was started. For example, the button label switches between **Attach to Project** and **Detach from Project** when the debugger is started from the Workbench **Project** tab but switches between **Attach to Mechanical** and **Detach from Mechanical** when the debugger is started from Mechanical. When attached, the debugger listens to the product executing the extension, pausing execution when the first breakpoint is hit so that you can examine the script. If you fail to attach the debugger to a product, no breakpoints are ever hit because the debugger is not listening for activity. When you detach, the debugger stops listening for activity.
	F5	Continue script execution. Clicking this button continues the execution of the script from the current breakpoint to either the next breakpoint or until the end of the script is reached.
	F10	Step over next function call. When the current line contains a function call, clicking this button resumes the debugger, which then pauses execution at the first line after the called function returns. If a breakpoint is set inside the called function, the debugger pauses at the breakpoint.
	F11	Step into next function call. When the current line contains a method statement, clicking this button moves the debugger from the current line into the method. Otherwise, it steps over the next function call.
	Shift + F11	Step out of current function. When you have stepped into a method or have a breakpoint inside a method, clicking this button steps you out of the method. Otherwise, it continues script execution.
	Ctrl + Shift + F10	Set next statement. Clicking this button jumps to the line where the cursor is currently positioned without executing any statements, allowing you to replay or skip lines.
	---	Clear breakpoints. Clicking this button removes all breakpoints (p. 173) from the scripts in loaded extensions.
	---	Clear watches. Clicking this button removes all watches (p. 179) on variables and expressions that you've set in the **Watch Expressions** tab.
Save Current File	---	Saves any changes that you've made to the current file and reloads the extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Navigating Scripts

Merely opening and attaching the debugger to an extension does not engage interactive debugging activity. You must also perform actions within the target product that call a loaded extension's script. Only then are breakpoints hit and execution details populated. For example, in the Workbench **Project** tab, to invoke the callback **`update()`** for the extension **`DebuggerDemo`**, you right-click the **Pre-Geometry** cell in the custom **Pre-Design** system and select **Update**.

The debugger listens as the script is executed. When it encounters the first breakpoint in `main.py`, the debugger pauses execution, highlighting the line in yellow to indicate that it is the next statement to execute.

In the debugger toolbar (p. 175), you have a button for stepping over to the next function call. Additionally, if the current breakpoint is for a method, you have buttons for stepping into the method call and then subsequently for stepping back out of the method call. You can also use the shortcut keys assigned to these buttons.

When execution of the extension is paused at a breakpoint, all functions, variables, and objects remain in memory. Using the various tabs in the debugger's lower panes, you can look at current values and review the call stack to determine if script violations or issues exist.

When you step over, into, or out of a function, the debugger automatically follows line execution. If you double-click a breakpoint, the debugger takes you to the script and line where this breakpoint is set. Similarly, when execution is paused, you can double-click an entry in the call stack (p. 176) to have the debugger take you to the script and corresponding line.

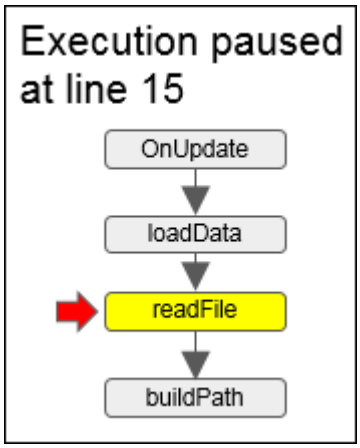
Using the Call Stack Tab

A call stack displays the sequence of executed function calls that have lead up to the currently paused statement. It is populated from the currently executing function, which is the most nested, to the top level, which is the shallowest function. For example, assume that a script has four functions and that a breakpoint is set at line 15:

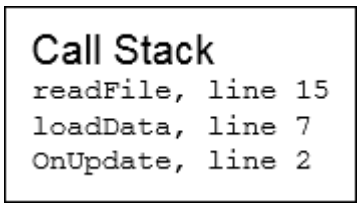
Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
1
2   def OnUpdate():
3       # some code
4       values = loadData(casename)
5       # some code
6
7   def loadData(name):
8       # some code
9       rawdata = readFile(filename)
10      # some code
11      return values
12
13  def readFile(filename):
14      # some code
15      fullPathName = buildPath(filename)
16      # some code
17      return rawData
18
19  def buildPath(filename):
20      # some code
21      return fullPathName
```

When the function **`OnUpdate()`** is executed, it calls the function **`LoadData()`**, which calls the function **`readFile()`**, which calls the function **`buildPath()`**. At the breakpoint set on line15, the debugger pauses while the function **`readFile()`** is executing.



At this point, the **Call Stack** tab displays these entries:

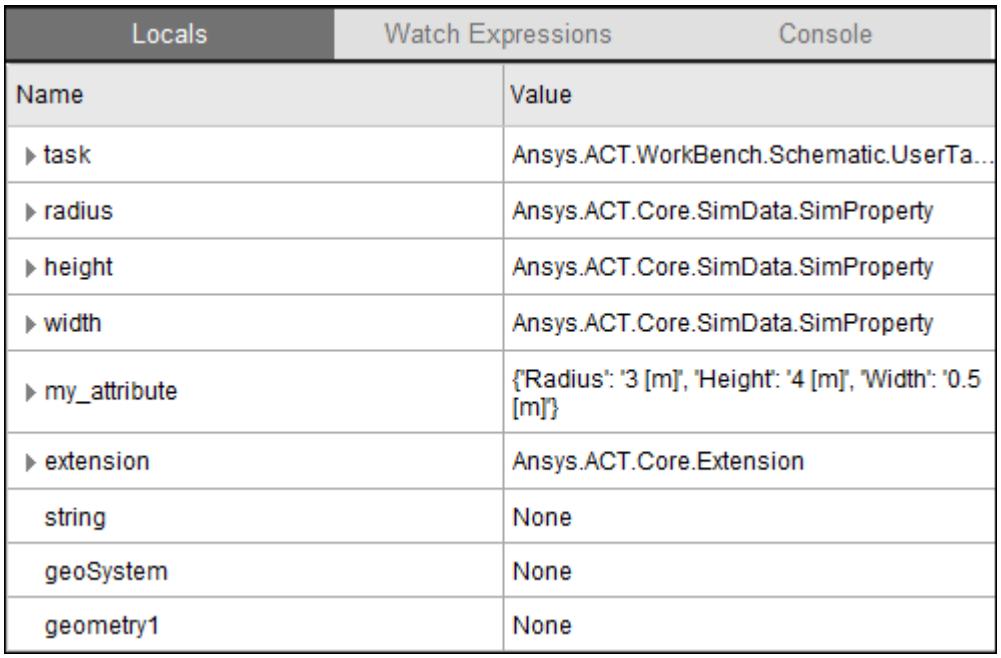


If you click an entry in the call stack, the debugger takes you to the corresponding script and line.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Using the Locals Tab

Locals refer to variables that are currently available for the active location in the script. In the **Locals** tab, you can see the values for all variables that are defined in the method, even if a statement that sets a variable value hasn't yet been executed. The variables are shown in the order in which they appear in the script.



In the **Locals** tab, you inspect variables to observe how values change as you step through subsequent statements. Clicking the arrow to the left of a variable switches between expanding and hiding its properties. By navigating through these variables, you can verify that they are storing the values that you expect at particular points in the execution of the script.

Using the Console Tab

For testing and manipulating scripts, you use the **Console** tab. This simplified version of the ACT Console (p. 53) allows you to execute commands and edit variables. For example, if the local variable **`extension`** has been set, you can type **`extension.Name`** in the command line and press the **Enter** key to return the name of the extension:

For the extension **`DebuggerDemo`**, assume that after completing the update to the **Pre-Geometry** cell, you inserted a breakpoint at line 50 of the **`script main.py`**. When you right-click this same cell in the **Project Schematic** and select **Edit**, the debugger listens as the extension executes, pausing execution at line 50.

In the **Console** tab, you can access the available local variables and change values. For example, typing **`system1`** and pressing the **Enter** key accesses your system:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Entering **`system1.HeaderText="My System"`** and then **`system1.DisplayText="MySystem"`** changes the system header and display text in the **Project Schematic** to **My System**.

As you execute commands in the console, the debugger reevaluates entries in both the **Locals** and **Watch Expressions** tabs.

Using the Watch Expressions Tab

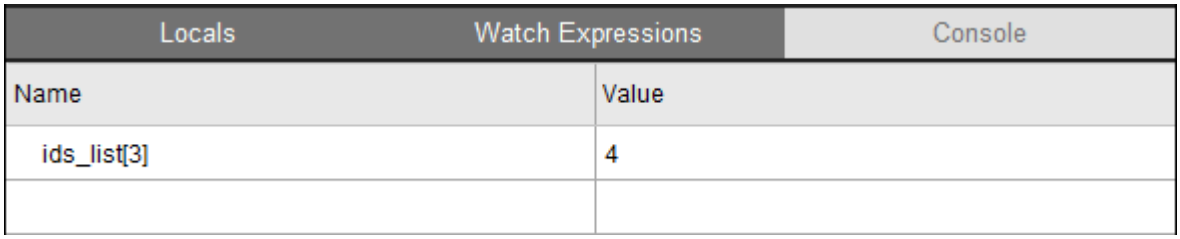
Watches refer to variables and expressions that you have chosen to observe and evaluate as the script executes. In the **Watch Expressions** tab, you add the specific variables or expressions that you want to watch during debugging by typing their names in the **Name** column of empty rows.

As you advance the debugger through the execution of the script, you use this tab to see how values change for the available watched variables. For the extension **`DebuggerDemo`**, assume that you have finished setting up the geometry in the **Project Schematic**. In the **Outline** view in Mechanical, you can right-click **Solution** in the tree and select **Insert** → **Debugger Result**.

To debug the script that Mechanical executes, you must first detach from the Workbench **Project** tab and close the debugger. You must then start the debugger from Mechanical and click the button in the debugger toolbar for attaching to Mechanical.

To have the debugger pause where the callback **`<evaluate>`** is adding the custom debugger result to the Mechanical tree, you can insert a breakpoint at the line where the result values are collected. In the **Outline** view in Mechanical, you then right-click **Solution** again and select **Evaluate All Results** to invoke the callback **`<evaluate>`**.

When the debugger pauses where the results values are collected, in the **Watch Expressions** tab, you can enter **`ids_list[3]`** in the **Name** column to see the index 3 value as the script executes:



To view the entire list of values for this IDs list, you can type the name for the variable in the **Name** column and then watch as the values populate:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Locals	Watch Expressions	Console
Name	Value	
ids_list[3]	4	
▸ ids_list	[1, 2, 3, 4]	

In the **Console** tab, you can directly manipulate this variable. After entering **`ids_list`** to access the list values, you can enter **`ids_list[3]=ids_list[3]*5`** to multiply the index 3 value by 5, changing it from 4 to 20. Entering **`ids_list`** again would allow you to see the changed value.

```
> ids_list[3]=ids_list[3]*5
> ids_list
< [1, 2, 3, 20]
```

In the **Watch Expressions** tab, the value for the index 3 value changes accordingly:

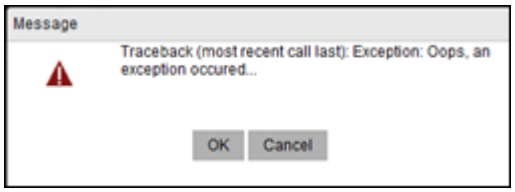
Locals	Watch Expressions	Console
Name	Value	
ids_list[3]	20	
▸ ids_list	[1, 2, 3, 20]	

You can navigate through the variables that you've added in the **Watch Expressions** tab and in some cases modify values in the **Value** column to see the effect. For example, for **`ids_list(3)`**, rather than programmatically changing the value in the **Console** tab, you can change the value for **`ids_list[3]`** directly to 20 in the **Value** column.

Any variables that cannot be set in the **Watch Expressions** tab are grayed out. If an expression has a syntax error, the **Value** column displays the same compiler error as you would see when entering this command in the console.

Handling Exceptions

Exceptions are error states that occur while a script is executed. When an exception is thrown, the debugger automatically stops and displays an error message, as shown in the following figure. When you click **OK**, you return to the debugger, where you can use the various tabs to debug the script and understand the exception. However, after an exception occurs, you cannot continue debugging the remaining statements. The script will finalize its execution without you having the ability to debug it.



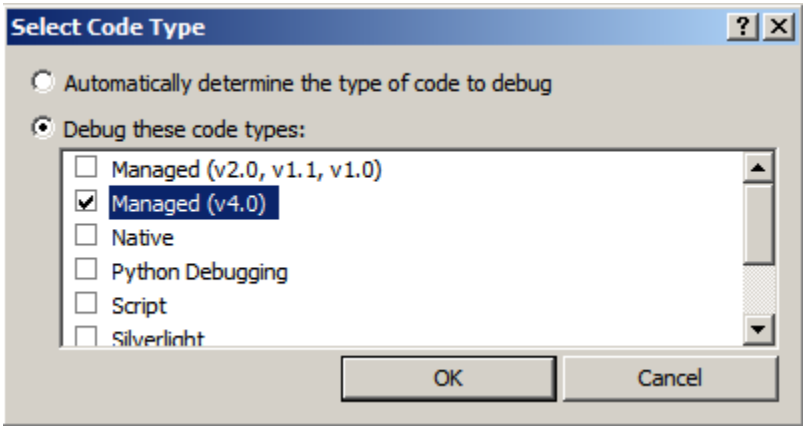
Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Debugging with Microsoft® Visual Studio

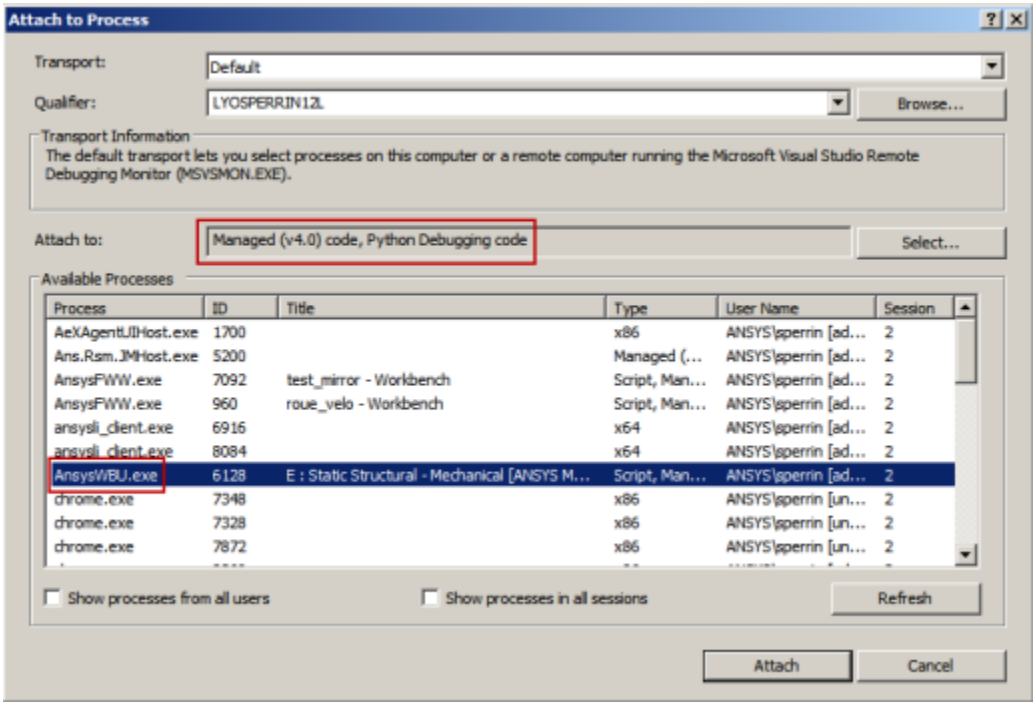
If you have Microsoft Visual Studio, you can use it to debug an IronPython script that is running.

For example, to debug an extension running in Mechanical:

1. Ensure that debug mode (p. 169) is enabled.
2. Start Visual Studio.
3. For the **Attach to** field, verify that the code type **Managed (v.4.0)** is selected.



4. Attach the process **`AnsysWBU.exe`**.



5. Open your IronPython script and set your breakpoints.

Refer to Microsoft Visual Studio documentation for information on using this program to debug your script.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.

ACT Customization Guides for Supported Ansys Products

Once you are familiar with the general ACT usage information in this guide, you can explore the ACT guides for customizing supported Ansys products:

- ACT Customization Guide for DesignModeler
- ACT Customization Guide for DesignXplorer
- ACT Customization Guide for Electronics Desktop
- ACT Customization Guide for Fluent
- ACT Customization Guide for Mechanical
- ACT Customization Guide for SpaceClaim
- ACT Customization Guide for Workbench

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.

Appendix A. Extension Elements

This appendix describes the basic XML elements used in an ACT extension. The XML file for an extension always begins with the element <extension> (p. 185). For comprehensive information on XML elements, see the *Ansys ACT XML Reference Guide*.

<extension>

The XML file has the element **<extension>** as the base tag or root node. It provides initialization and configuration information for the extension. All other elements fall under the element **<extension>**.

```
<extension version="[version number]" minorversion="[minor version number]" name="[extension name]">
          <author>author name</author>
          <description>description of the extension</description>
          <assembly src="[source file name (string)]" namespace="[namespace (string)]"/>
</extension>
```

Descriptions follow of the secondary elements and attributes for the element **<extension>**. Click the links to view the corresponding sections in this appendix.

Secondary XML elements under the element <extension>

<application> (p. 187)

Defines a new ACT app or extension.

<appstoreid> (p. 188)

Defines a unique identifier for the ACT app to use in the Ansys Store.

<assembly> (p. 188)

Defines the assembly to be loaded.

<author> (p. 189)

Defines the author of the extension.

<description> (p. 189)

Defines the description of the extension.

<guid> (p. 189)

Defines a unique identifier for the extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

<interface> (p. 190)

Specifies the customizations to be done at the interface level.

<licenses> (p. 193)

Defines a licenses collection for the extension.

<script> (p. 193)

Specifies the IronPython script that defines the functions called by the extension.

<simdata> (p. 193)

Defines a general section that stores all user object definitions and specifies the custom features to be integrated.

<templates> (p. 196)

Defines a collection of control templates.

<uidefintion> (p. 196)

Defines the user interface (customized panels and layouts) for the extension.

<wizard> (p. 196)

Defines one or more wizards within the extension.

<workflow> (p. 198)

Defines custom workflows composed of process integration items (tasks and task groups).

Attributes for the element <extension>

`version`

Major version of the extension.

Mandatory attribute.

```
version="[version number (integer)]"
```

`name`

Name of the extension.

Mandatory attribute.

```
name="[extension name (string)]"
```

`minorversion`

Minor version of the extension.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Optional attribute.

```
minorversion="[minor version number (integer)]"
```

debug

Specifies if the scripted version of the extension should be opened in the debug mode.

Optional attribute.

icon

Icon for the extension.

Optional attribute.

<application>

Defines a new application.

```
<application>
 <callbacks> ... </callbacks>
 <description>
 <description>
 <panel>
</application>
```

Child tags for the element <application>

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script.

<description>

Description of the application.

<Panel>

Defines a panel to display.

Attributes for the element <application>

name

Name of the application.

Mandatory.

class

Class name of the controller of the application.

Optional.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

`context`

Defines a context or combination of contexts (separated using '|') in which the application can be launched.

Optional.

`MainPanel`

Name of the first panel to display.

Optional.

Callbacks for the element <application>

`<OnApplicationFinished>`

Invoked when the application finishes.

`<OnApplicationInitialized>`

Invoked when the application initializes.

`<OnApplicationStarted>`

Invoked when the application starts.

<appstoreid>

Defines the unique identifier to use in the Ansys Store. This tag must be written exclusively in lowercase.

No child tags, attributes, or callbacks.

```
<appstoreid>appstoreid</appstoreid>
```

<assembly>

Defines the assembly to be loaded.

```
<assembly src="[file name]" namespace="[namespace]"/>
```

Attributes for the element <assembly>

`context`

Context or combination of contexts (separated using '|') for the import.

Mandatory.

`namespace`

Namespace to import.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Mandatory.

```
namespace="[namespace]"
```

src

Name of the DLL file to import.

Mandatory.

```
src="[file name]"
```

<author>

Defines the author of the extension.

No child tags, attributes, or callbacks.

```
<author>[Name of the author or organisation (string)]</author>
```

<description>

Defines the description of the extension.

No child tags, attributes, or callbacks.

```
<description>[Description (string)]</description>
```

<guid>

Defines the GUID (global unique identifier) for the extension.

```
<guid shortid="[name (string)]">GUID</guid>
```

A GUID ensures that two different extensions with the same name are never in conflict. A GUID must be added before the first build of a binary extension and must never changed. When you update an extension, add features to an extension, or create a new version of the extension, never change the GUID.

Because ACT considers two extensions with the same GUID as the same extension, you can change the extension name in a new version of the extension without compromising compatibility with projects that were saved with the older version of the extension.

Attributes for the element <guid>

shortid

Short identifier for backward compatibility. Must be the same as the extension name for all extensions created before R15.

Optional.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
shortid="[extension name (string)]"
```

<interface>

Defines the user interface for the extension.

```
<extension version="[version id (integer)]" name="[extension name (string)]"
  <interface context="[Project | Mechanical]">
   ...
  </interface>
```

Child tags for the element <interface>

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script.

<filter>

Defines a filter.

<images>

Defines the default folder where images to be used by the extension are stored.

```
<images>[folder]</images>
```

<toolbar>

Defines a toolbar.

```
 <toolbar name="[toolbar internal name (string)]" caption="[toolbar display name (string)]">
    <entry>...</entry>
 </toolbar>
```

Attributes for the element <interface>

context

Context or combination of contexts (separated using '|') for the interface.

Mandatory.

```
context="[context name]”
```

Callbacks for the element <interface>

<GetCommands>

Called to collect commands to add to the solver input file. The attribute **location** indicates where the commands are to be inserted. The attribute **location** can be set to any of these values: **init**, **pre**, **post**, **solve**, and **preload**:

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
<getcommands> location="[init | pre | post | solve | preload">[function(analysis,stream)]</getcommands>
```

<IsAnalysisValid>

Called to check if an analysis is valid.

```
<isanalysisvalid>[function(solver)]</isanalysisvalid>
```

<OnActiveObjectChange>

Called when the active object is changed.

<OnAfterGeometryUpdate>

Called after the geometry has been updated.

<OnAfterRemove>

Called after the object has been removed.

<OnAfterSolve>

Called after an analysis has been solved.

```
<onaftersolve>[function(analysis)]</onaftersolve>
```

<OnBeforeGeometryUpdate>

Called before the geometry is starts to update.

<OnBeforeSolve>

Called before an analysis starts to solve.

```
<onbeforesolve>[function(analysis)]</onbeforesolve>
```

<OnBodySuppressStateChange>

Called when the body suppress state has been changed.

<OnDraw>

Called when the application is drawn.

<OnDraw2D>

Called when the application is drawn.

```
<ondraw>[function()]</ondraw>
```

<OnInit>

Called when the given context is initialized.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
<oninit>[function name(application context)] </oninit>
```

<OnLoad>

Called when a project is loaded.

```
<onload>[function(currentFolder)]</onload>
```

<OnMeshCleaned>

Called when the mesh is cleaned.

<OnMeshGenerated>

Called when the mesh is generated.

<OnPostFinished>

Called when the postprocessing ends for a given analysis.

```
<onpostfinished>[function(analysis)]</onpostfinished>
```

<OnPostStarted>

Called when the postprocessing starts for a given analysis.

```
<onpoststarted>[function(analysis)]</onpoststarted>
```

<OnReady>

Called when the application is fully loaded and in a "ready" state.

<OnSave>

Called when the project is saved.

```
<onsave>[function(currentFolder)]</onsave>
```

<OnTerminate>

Called when the given context is terminated.

```
<onterminate>[function(context)]</onterminate>
```

<Resume>

Called when a project is loaded.

```
<resume>[function(binary reader)]</resume>
```

<Save>

Called when a project is saved.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
<save>[function(binary writer)]</save>
```

<licenses>

Defines a licenses collection for the extension.

No child tags, attributes, or callbacks.

<script>

Specifies the IronPython script referenced by the extension.

```
<extension version="[version id (integer)]" name="[extension name (string)]"
  <script src="[python file name (string)]"></script></extension>
```

You can either insert the IronPython script directly into the XML extension definition file or use the attribute **`src`** to specify the path to the script.

An additional script can be specified by adding a new element **`<script>`**. For example:

```
<script src="[Path]\filename.py" />
```

If the **`src`** attribute is defined, then the tag content is ignored.

By default, ACT looks for IronPython scripts in the same directory as the extension. If the scripts are not located in that directory, you can specify the path the scripts in addition to the file name. For example:

```
<script src="my_path\main.py" />
```

Attributes for the element <script>

`compiled`

Specifies whether the script is to be compiled as a binary file.

Optional.

```
compiled="[false(default) | true]"
```

`src`

Specifies the IronPython script referenced by the extension.

Optional.

```
src="[python file name (string)]"
```

<simdata>

Defines a general section that stores all user object definitions.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

The element **<simdata>** information pertains specifically to the simulation environment. Child elements are used for integrating custom simulation features into the application. These main features are nested as child elements within the **<simdata>** element.

```
<simdata>
 <load>
 <object>
 <optimizer>
 <solver>
 <geometry>
 <result>
 <step>
 <ExtensionObject>
 <Sampling>
</simdata>
```

Child tags for the element <simdata>

<load>

Defines a simulation load or boundary.

```
<load name="[load internal name]"
  version="[version identifier of the load]"
  caption="[load display name]"
  icon="[name of an icon file]"
  issupport="[true | false]"
  isload="[true | false]"
  color="[#xxxxxx]"
  contextual="[true(default) | false]"
  class="[class name]"
  unit="[Default unit name]"
  ...
</load>
```

<object>

Defines a simulation object.

```
<object>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
 <target>
</object>
```

<optimizer>

Defines an optimizer.

```
<optimizer>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
</optimizer>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

<solver>

Specifies a third-party solver to be used in the simulation.

```
<solver>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
</solver>
```

<geometry>

Defines a geometry feature.

```
<geometry>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
</geometry>
```

<result>

Defines a custom result.

```
<result>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
</result>
```

<step>

Defines a step in a wizard.

```
<step>
 <callbacks> ... </callbacks>
 <description>
 <property>
 <propertygroup>
</step>
```

<ExtensionObject>

Extends the extension object definition. (Inherited from DesignXplorer **SimEntity**)

```
<extensionobject>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
 <target>
</extensionobject>
```

<Sampling>

Defines a custom sampling.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
<sampling>
 <callbacks> ... </callbacks>
 <property>
 <propertygroup>
 <propertytable>
 <target>
</sampling>
```

Attributes for the element <simdata>

`context`

Context or combination of contexts (separated using '|').

Mandatory.

```
context="[Project | targetproduct | targetproduct]">
```

<templates>

Defines a collection of control templates.

```
  <templates>
    <controltemplate name="[template name (string)]" version="[version id (integer)]">
      <propertygroup>
        <property> ... </property>
        <propertygroup> ... </propertygroup>
      </propertygroup>
    </controltemplate>
  </templates>
```

No child tags, attributes, or callbacks.

<uidefintion>

Defines one or more layouts that can be used for wizards.

```
<uidefinition>
 <layout>
</uidefinition>
```

Child tags for the element <uidefintion>

`<ControlTemplate>`

Defines a control template for the creation of groups of properties.

<wizard>

Defines one or more wizards within the extension.

```
<wizard>
 <author>
 <description>
```


Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

```
 <step></step>
</wizard>
```

Child tags for the element <wizard>

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script.

Attributes for the element <wizard>

<context>

Context or combination of contexts (separated using '|') for the wizard.

Mandatory.

<name>

Name of the wizard.

Mandatory.

<version>

Version of the wizard.

Mandatory.

<caption>

Caption for the wizard.

Optional.

<icon>

Icon for the wizard.

Optional.

<layout>

Layout of the wizard.

Optional.

<description>

Description of the wizard.

Optional.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Callbacks for the element <wizard>

<canstart>

Invoked to determine whether the wizard can be started.

<workflow>

Defines custom workflows composed of process integration items (tasks and task groups). Defines the top-level workflow tag within an ACT app.

```
<workflow>
 <callbacks> ... </callbacks>
 <taskgroups>
 <tasks>
</workflow>
```

Child tags for the element <workflow>

<Callbacks>

Specifies the callbacks that invoke functions from the IronPython extension script. (Inherited from **SimEntity**)

<TaskGroups>

Task groupings to be exposed as organized blocks within the workflow.

<Tasks>

Tasks exposed by this workflow.

Attributes for the element <workflow>

caption

Caption for the object. (Inherited from **SimEntity**)

Optional.

class

Class name of the controller of the object. (Inherited from **SimEntity**)

Optional.

context

Context (application) to which this workflow applies.

Mandatory.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

contextual

Indicates whether the object must be displayed in the contextual menu. (Inherited from **SimEntity**)

Optional.

icon

Icon for the object. (Inherited from **SimEntity**)

Optional.

name

Name of the object. (Inherited from **SimEntity**)

Mandatory.

version

Version of the object. (Inherited from **SimEntity**)

Mandatory.

Callbacks for the element <workflow>

<onbeforedesignpointchanged>

Invoked before the design point changes.

<onafterdesignpointchanged>

Invoked after the design point changes.

<onbeforetaskreset>

Invoked before the task is reset back to its pristine, new state.

<onaftertaskreset>

Invoked after the task has been reset back to its pristine, new state.

<onbeforetaskrefresh>

Invoked before the task consumes all upstream data and prepares any local data for an ensuing update.

<onaftertaskrefresh>

Invoked after the task has consumed all upstream data and has prepared any local data for an ensuing update.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

<onbeforetaskupdate>

Invoked before the task generates all broadcast output types that render the component fully solved.

<onaftertaskupdate>

Invoked after the task has generated all broadcast output types that render the component fully solved.

<onbeforetaskcreate>

Invoked before the task is created based on an underlying template.

<onaftertaskcreate>

Invoked after the task has been created based on an underlying template.

<onbeforetaskdelete>

Invoked before the task is removed from a task group.

<onaftertaskdelete>

Invoked after the task has been removed from a task group.

<onbeforetaskduplicate>

Invoked before an identical, yet independent, clone of the task is created.

<onaftertaskduplicate>

Invoked after an identical, yet independent, clone of the task has been created.

<onbeforetasksourceschanged>

Invoked before the task processes a change in upstream sources.

<onaftertasksourceschanged>

Invoked after the task has processed a change in upstream sources.

<onbeforetaskcanusertransfer>

Invoked before the task checks whether it can consume data from a specific upstream task.

<onaftertaskcanusertransfer>

Invoked after the task has checked whether it can consume data from a specific upstream task.

<onbeforetaskcanduplicate>

Invoked before the task checks whether it permits duplication.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

<onaftertaskcanduplicate>

Invoked after the task has checked whether it permits duplication.

<onbeforetaskstatus>

Invoked before the task calculates its current state.

<onaftertaskstatus>

Invoked after the task has calculated its current state.

<onbeforetaskpropertyretrieval>

Invoked before the task determines the visibility of its property-containing objects.

<onaftertaskpropertyretrieval>

Invoked after the task has determined the visibility of its property-containing objects.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information of ANSYS, Inc. and its subsidiaries and affiliates.

Release 2022 R2 - © ANSYS, Inc. All rights reserved. - Contains proprietary and confidential information
of ANSYS, Inc. and its subsidiaries and affiliates.