PyAnsys Geometry

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

Oct 24, 2023

CONTENTS

PyAnsys Geometry is a Python client library for the Ansys Geometry service.

Getting started   Learn how to run the Windows Docker container, install the PyAnsys Geometry image, and launch and connect to the Geometry service.

Getting started

User guide   Understand key concepts and approaches for primitives, sketches, and model designs.

User guide

API reference   Understand PyAnsys Geometry API endpoints, their capabilities, and how to interact with them programmatically.

API reference

Examples   Explore examples that show how to use PyAnsys Geometry to perform many different types of operations.

Examples

Contribute   Learn how to contribute to the PyAnsys Geometry codebase or documentation.

Contribute

Assets   Download different assets related to PyAnsys Geometry, such as documentation, package wheelhouse, and related files.

assets.html

Assets

CHAPTER

ONE

GETTING STARTED

PyAnsys Geometry is a Python client library for the Ansys Geometry service.

1.1 Available modes

This client library works with a Geometry service backend. There are several ways of running this backend, although the preferred and high-performance mode is using Docker containers. Select the option that suits your needs best.

Docker containers   Launch the Geometry service as a Docker container and connect to it from PyAnsys Geometry.

Local service   Launch the Geometry service locally on your machine and connect to it from PyAnsys Geometry.

Remote service   Launch the Geometry service on a remote machine and connect to it using PIM (Product Instance Manager).

Connect to an existing service   Connect to an existing Geometry service locally or remotely.

1.2 Compatibility with Ansys releases

PyAnsys Geometry continues to evolve as the Ansys products move forward. For more information, see *Ansys product version compatibility*.

1.3 Development installation

In case you want to support the development of PyAnsys Geometry, install the repository in development mode. For more information, see *Install package in development mode*.

1.4 Frequently asked questions

Any questions? Refer to *Q&A* before submitting an issue.

1.4.1 Docker containers

What is Docker?

Docker is an open platform for developing, shipping, and running apps in a containerized way.

Containers are standard units of software that package the code and all its dependencies so that the app runs quickly and reliably from one computing environment to another.

Ensure that the machine where the Geometry service is to run has Docker installed. Otherwise, see Install Docker Engine in the Docker documentation.

Select your Docker container

Currently, the Geometry service backend is mainly delivered as a **Windows** Docker container. However, these containers require a Windows machine to run them.

A Linux version of the Geometry service is also available but with limited capabilities, meaning that certain operations are not available or fail.

Select the kind of Docker container you want to build:

Windows Docker container Build a Windows Docker container for the Geometry service and use it from PyAnsys Geometry. Explore the full potential of the Geometry service.

Linux Docker container Test out the Linux Docker container for the Geometry service, which has limited functionalities.

Go to Getting started

Windows Docker container

Contents

Docker for Windows containers

To run the Windows Docker container for the Geometry service, ensure that you follow these steps when installing Docker:

1. Install Docker Desktop.
2. When prompted for **Use WSL2 instead of Hyper-V (recommended)**, **clear** this checkbox. Hyper-V must be enabled to run Windows Docker containers.
3. Once the installation finishes, restart your machine and start Docker Desktop.
4. On the Windows taskbar, go to the **Show hidden icons** section, right-click in the Docker Desktop app, and select **Switch to Windows containers**.

Now that your Docker engine supports running Windows Docker containers, you can build or install the PyAnsys Geometry image.

Build or install the Geometry service image

There are two options for installing the PyAnsys Geometry image:

- Download it from the *GitHub Container Registry*.
- *Build the Geometry service Windows container*.

GitHub Container Registry

Note: This option is only available for users with write access to the repository or who are members of the Ansys organization.

Once Docker is installed on your machine, follow these steps to download the Windows Docker container for the Geometry service and install this image.

1. Using your GitHub credentials, download the Docker image from the PyAnsys Geometry repository on GitHub.
2. Use a GitHub personal access token with permission for reading packages to authorize Docker to access this repository. For more information, see Managing your personal access tokens in the GitHub documentation.
3. Save the token to a file with this command:

```
echo XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX > GH_TOKEN.txt
```

4. Authorize Docker to access the repository and run the commands for your OS. To see these commands, click the tab for your OS.

Powershell

```
$env:GH_USERNAME=<my-github-username>
cat GH_TOKEN.txt | docker login ghcr.io -u $env:GH_USERNAME --password-stdin
```

Windows CMD

```
SET GH_USERNAME=<my-github-username>
type GH_TOKEN.txt | docker login ghcr.io -u %GH_USERNAME% --password-stdin
```

5. Pull the Geometry service locally using Docker with a command like this:

```
docker pull ghcr.io/ansys/geometry:windows-latest
```

Build the Geometry service Windows container

The Geometry service Docker containers can be easily built by following these steps.

Inside the repository's `docker` folder, there are two `Dockerfile` files:

- `Dockerfile.linux`: Builds the Linux-based Docker image.
- `Dockerfile.windows`: Builds the Windows-based Docker image.

Depending on the characteristics of the Docker engine installed on your machine, either one or the other has to be built.

This guide focuses on building the `Dockerfile.windows` image.

Prerequisites

- Ensure that Docker is installed in your machine. If you do not have Docker available, see *Docker for Windows containers*.
- Download the latest Windows Dockerfile.
- Download the latest release artifacts for the Windows Docker container (ZIP file) for your version.

Note: Only users with access to https://github.com/ansys/pyansys-geometry-binaries can download these binaries.

- Move this ZIP file to the location of the Windows Dockerfile previously downloaded.

Build the Docker image

To build your image, follow these instructions:

1. Navigate to the folder where the ZIP file and Dockerfile are located.
2. Run this Docker command:

```
docker build -t ghcr.io/ansys/geometry:windows-latest -f Dockerfile.windows .
```

3. Check that the image has been created successfully. You should see output similar to this:

```
docker images

>>> REPOSITORY                                              TAG                 ␣
↪             IMAGE ID       CREATED          SIZE
>>> ghcr.io/ansys/geometry                                  windows-******      ␣
↪             ............   X seconds ago    Y.ZZGB
>>> ......                                                  ......              ␣
↪             ............   ..............   ......
```

Launch the Geometry service

There are methods for launching the Geometry service:

- You can use the PyAnsys Geometry launcher.
- You can manually launch the Geometry service.

Environment variables

The Geometry service requires this mandatory environment variable for its use:

- `LICENSE_SERVER`: License server (IP address or DNS) that the Geometry service is to connect to. For example, `127.0.0.1`.

You can also specify other optional environment variables:

- `ENABLE_TRACE`: Whether to set up the trace level for debugging purposes. The default is `0`, in which case the trace level is not set up. Options are `1` and `0`.
- `LOG_LEVEL`: Sets the Geometry service logging level. The default is `2`, in which case the logging level is `INFO`.

Here are some terms to keep in mind:

- **host**: Machine that hosts the Geometry service. It is typically on `localhost`, but if you are deploying the service on a remote machine, you must pass in this host machine's IP address when connecting. By default, PyAnsys Geometry assumes it is on `localhost`.
- **port**: Port that exposes the Geometry service on the host machine. Its value is assumed to be `50051`, but users can deploy the service on preferred ports.

Prior to using the PyAnsys Geometry launcher to launch the Geometry service, you must define general environment variables required for your OS. You do not need to define these environment variables prior to manually launching the Geometry service.

Using PyAnsys Geometry launcher

Define the following general environment variables prior to using the PyAnsys Geometry launcher. Click the tab for your OS to see the appropriate commands.

Linux/Mac

```
export ANSRV_GEO_LICENSE_SERVER=127.0.0.1
export ANSRV_GEO_ENABLE_TRACE=0
export ANSRV_GEO_LOG_LEVEL=2
export ANSRV_GEO_HOST=127.0.0.1
export ANSRV_GEO_PORT=50051
```

Powershell

```
$env:ANSRV_GEO_LICENSE_SERVER="127.0.0.1"
$env:ANSRV_GEO_ENABLE_TRACE=0
$env:ANSRV_GEO_LOG_LEVEL=2
$env:ANSRV_GEO_HOST="127.0.0.1"
$env:ANSRV_GEO_PORT=50051
```

Windows CMD

```
SET ANSRV_GEO_LICENSE_SERVER=127.0.0.1
SET ANSRV_GEO_ENABLE_TRACE=0
SET ANSRV_GEO_LOG_LEVEL=2
SET ANSRV_GEO_HOST=127.0.0.1
SET ANSRV_GEO_PORT=50051
```

Warning: When running a Windows Docker container, certain high-value ports might be restricted from its use. This means that the port exposed by the container has to be set to lower values. You should change the value of `ANSRV_GEO_PORT` to use a port such as `700`, instead of `50051`.

Manual launch

You do not need to define general environment variables prior to manually launching the Geometry service. They are directly passed to the Docker container itself.

Geometry service launcher

As mentioned earlier, you can launch the Geometry service locally in two different ways. To see the commands for each method, click the following tabs.

Using PyAnsys Geometry launcher

This method directly launches the Geometry service and provides a `Modeler` object.

```python
from ansys.geometry.core.connection import launch_modeler

modeler = launch_modeler()
```

The `launch_modeler()` method launches the Geometry service under the default conditions. For more configurability, use the `launch_local_modeler()` method.

Manual launch

This method requires that you manually launch the Geometry service. Remember to pass in the different environment variables that are needed. Afterwards, see the next section to understand how to connect to this service instance from PyAnsys Geometry.

Linux/Mac

```bash
docker run \
    --name ans_geo \
    -e LICENSE_SERVER=<LICENSE_SERVER> \
    -p 50051:50051 \
    ghcr.io/ansys/geometry:<TAG>
```

Powershell

```powershell
docker run `
    --name ans_geo `
    -e LICENSE_SERVER=<LICENSE_SERVER> `
    -p 50051:50051 `
    ghcr.io/ansys/geometry:<TAG>
```

Windows CMD

```bat
docker run ^
    --name ans_geo ^
    -e LICENSE_SERVER=<LICENSE_SERVER> ^
    -p 50051:50051 ^
    ghcr.io/ansys/geometry:<TAG>
```

Warning: When running a Windows Docker container, certain high-value ports might be restricted from its use. This means that the port exposed by the container has to be set to lower values. You should change the value of `-p 50051:50051` to use a port such as `-p 700:50051`.

Connect to the Geometry service

After the Geometry service is launched, connect to it with these commands:

```python
from ansys.geometry.core import Modeler

modeler = Modeler()
```

By default, the `Modeler` instance connects to `127.0.0.1` (`"localhost"`) on port `50051`. You can change this by modifying the `host` and `port` parameters of the `Modeler` object, but note that you must also modify your `docker run` command by changing the `<HOST-PORT>-50051` argument.

The following tabs show the commands that set the environment variables and `Modeler` function.

> **Warning:** When running a Windows Docker container, certain high-value ports might be restricted from its use. This means that the port exposed by the container has to be set to lower values. You should change the value of `ANSRV_GEO_PORT` to use a port such as `700`, instead of `50051`.

Environment variables

Linux/Mac

```
export ANSRV_GEO_HOST=127.0.0.1
export ANSRV_GEO_PORT=50051
```

Powershell

```
$env:ANSRV_GEO_HOST="127.0.0.1"
$env:ANSRV_GEO_PORT=50051
```

Windows CMD

```
SET ANSRV_GEO_HOST=127.0.0.1
SET ANSRV_GEO_PORT=50051
```

Modeler function

```python
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler(host="127.0.0.1", port=50051)
```

Go to Docker containers

Go to Getting started

Linux Docker container

Contents

Docker for Linux containers

To run the Linux Docker container for the Geometry service, ensure that you follow these steps when installing Docker:

Linux machines

If you are on a Linux machine, install Docker for your distribution

Windows/MacOS machines

1. Install Docker Desktop for Windows or Docker Desktop for MacOS.
2. **(On Windows)** When prompted for **Use WSL2 instead of Hyper-V (recommended)**, **clear** this checkbox. Hyper-V must be enabled to run Windows Docker containers, which you might be interested in doing in the future.
3. Once the installation finishes, restart your machine and start Docker Desktop.

Now that your Docker engine supports running Linux Docker containers, you can build or install the PyAnsys Geometry image.

Build or install the Geometry service image

There are two options for installing the PyAnsys Geometry image:

- Downloading it from the *GitHub Container Registry*.
- *Build the Geometry service Linux container*.

GitHub Container Registry

Note: This option is only available for users with write access to the repository or who are members of the Ansys organization.

Once Docker is installed on your machine, follow these steps to download the Linux Docker container for the Geometry service and install this image.

1. Using your GitHub credentials, download the Docker image from the PyAnsys Geometry repository on GitHub.
2. Use a GitHub personal access token with permission for reading packages to authorize Docker to access this repository. For more information, see Managing your personal access tokens in the GitHub documentation.
3. Save the token to a file with this command:

```
echo XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX > GH_TOKEN.txt
```

4. Authorize Docker to access the repository and then run the commands for your OS. To see these commands, click the tab for your OS.

Linux/Mac

```
GH_USERNAME=<my-github-username>
cat GH_TOKEN.txt | docker login ghcr.io -u $GH_USERNAME --password-stdin
```

Powershell

```
$env:GH_USERNAME=<my-github-username>
cat GH_TOKEN.txt | docker login ghcr.io -u $env:GH_USERNAME --password-stdin
```

Windows CMD

```
SET GH_USERNAME=<my-github-username>
type GH_TOKEN.txt | docker login ghcr.io -u %GH_USERNAME% --password-stdin
```

5. Pull the Geometry service locally using Docker with a command like this:

```
docker pull ghcr.io/ansys/geometry:linux-latest
```

Build the Geometry service Linux container

The Geometry service Docker containers can be easily built by following these steps.

Inside the repository's `docker` folder, there are two `Dockerfile` files:

- `Dockerfile.linux`: File for building the Linux-based Docker image.
- `Dockerfile.windows`: File for building the Windows-based Docker image.

Depending on the characteristics of the Docker engine installed on your machine, either one or the other has to be built.

This guide focuses on building the `Dockerfile.linux` image.

Prerequisites

- Ensure that Docker is installed on your machine. If you do not have Docker available, see *Docker for Linux containers*.
- Download the latest Linux Dockerfile.
- Download the latest release artifacts for the Linux Docker container (ZIP file) according to your version.

Note: Only users with access to https://github.com/ansys/pyansys-geometry-binaries can download these binaries.

- Move this ZIP file to the location of the Linux Dockerfile previously downloaded.

Build the Docker image

To build your image, follow these steps:

1. Navigate to the folder where the ZIP file and the Dockerfile are located.
2. Run this Docker command:

```
docker build -t ghcr.io/ansys/geometry:linux-latest -f Dockerfile.linux .
```

3. Check that the image has been created successfully. You should see an output similar to this one:

```
docker images

>>> REPOSITORY                                                  TAG                ␣
↪             IMAGE ID       CREATED          SIZE
>>> ghcr.io/ansys/geometry                                      linux-******       ␣
↪             ............   X seconds ago    Y.ZZGB
>>> ......                                                      ......             ␣
↪             ............   ..............   ......
```

Launch the Geometry service

There are methods for launching the Geometry service:

- You can use the PyAnsys Geometry launcher.
- You can manually launch the Geometry service.

Environment variables

The Geometry service requires this mandatory environment variable for its use:

- `LICENSE_SERVER`: License server (IP address or DNS) that the Geometry service is to connect to. For example, `127.0.0.1`.

You can also specify other optional environment variables:

- `ENABLE_TRACE`: Whether to set up the trace level for debugging purposes. The default is `0`, in which case the trace level is not set up. Options are `1` and `0`.
- `LOG_LEVEL`: Sets the Geometry service logging level. The default is `2`, in which case the logging level is `INFO`.

Here are some terms to keep in mind:

- **host**: Machine that hosts the Geometry service. It is typically on `localhost`, but if you are deploying the service on a remote machine, you must pass in this host machine's IP address when connecting. By default, PyAnsys Geometry assumes it is on `localhost`.
- **port**: Port that exposes the Geometry service on the host machine. Its value is assumed to be `50051`, but users can deploy the service on preferred ports.

Prior to using the PyAnsys Geometry launcher to launch the Geometry service, you must define general environment variables required for your OS. You do not need to define these environment variables prior to manually launching the Geometry service.

Using PyAnsys Geometry launcher

Define the following general environment variables prior to using the PyAnsys Geometry launcher. Click the tab for your OS to see the appropriate commands.

Linux/Mac

```
export ANSRV_GEO_LICENSE_SERVER=127.0.0.1
export ANSRV_GEO_ENABLE_TRACE=0
export ANSRV_GEO_LOG_LEVEL=2
export ANSRV_GEO_HOST=127.0.0.1
export ANSRV_GEO_PORT=50051
```

Powershell

```
$env:ANSRV_GEO_LICENSE_SERVER="127.0.0.1"
$env:ANSRV_GEO_ENABLE_TRACE=0
$env:ANSRV_GEO_LOG_LEVEL=2
$env:ANSRV_GEO_HOST="127.0.0.1"
$env:ANSRV_GEO_PORT=50051
```

Windows CMD

```
SET ANSRV_GEO_LICENSE_SERVER=127.0.0.1
SET ANSRV_GEO_ENABLE_TRACE=0
SET ANSRV_GEO_LOG_LEVEL=2
SET ANSRV_GEO_HOST=127.0.0.1
SET ANSRV_GEO_PORT=50051
```

Manual launch

You do not need to define general environment variables prior to manually launching the Geometry service. They are directly passed to the Docker container itself.

Geometry service launcher

As mentioned earlier, you can launch the Geometry service locally in two different ways. To see the commands for each method, click the following tabs.

Using PyAnsys Geometry launcher

This method directly launches the Geometry service and provides a `Modeler` object.

```
from ansys.geometry.core.connection import launch_modeler

modeler = launch_modeler()
```

The `launch_modeler()` method launches the Geometry service under the default conditions. For more configurability, use the `launch_local_modeler()` method.

Manual launch

This method requires that you manually launch the Geometry service. Remember to pass in the different environment variables that are needed. Afterwards, see the next section to understand how to connect to this service instance from PyAnsys Geometry.

Linux/Mac

```
docker run \
    --name ans_geo \
    -e LICENSE_SERVER=<LICENSE_SERVER> \
    -p 50051:50051 \
    ghcr.io/ansys/geometry:<TAG>
```

Powershell

```
docker run `
    --name ans_geo `
    -e LICENSE_SERVER=<LICENSE_SERVER> `
    -p 50051:50051 `
    ghcr.io/ansys/geometry:<TAG>
```

Windows CMD

```
docker run ^
    --name ans_geo ^
    -e LICENSE_SERVER=<LICENSE_SERVER> ^
    -p 50051:50051 ^
    ghcr.io/ansys/geometry:<TAG>
```

Connect to the Geometry service

After the Geometry service is launched, connect to it with these commands:

```
from ansys.geometry.core import Modeler

modeler = Modeler()
```

By default, the `Modeler` instance connects to `127.0.0.1` (`"localhost"`) on port `50051`. You can change this by modifying the `host` and `port` parameters of the `Modeler` object, but note that you must also modify your `docker run` command by changing the `<HOST-PORT>-50051` argument.

The following tabs show the commands that set the environment variables and `Modeler` function.

Environment variables

Linux/Mac

```
export ANSRV_GEO_HOST=127.0.0.1
export ANSRV_GEO_PORT=50051
```

Powershell

```
$env:ANSRV_GEO_HOST="127.0.0.1"
$env:ANSRV_GEO_PORT=50051
```

Windows CMD

```
SET ANSRV_GEO_HOST=127.0.0.1
SET ANSRV_GEO_PORT=50051
```

Modeler function

```
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler(host="127.0.0.1", port=50051)
```

Go to Docker containers

Go to Getting started

1.4.2 Launch a local session

If Ansys 2023 R2 or later and PyAnsys Geometry are installed, you can create a local backend session using Discovery, SpaceClaim, or the Geometry service. Once the backend is running, PyAnsys Geometry can manage the connection.

To launch and establish a connection to the service, open Python and use the following commands for either Discovery, SpaceClaim, or the Geometry service.

Discovery

```
from ansys.geometry.core import launch_modeler_with_discovery

modeler_discovery = launch_modeler_with_discovery()
```

SpaceClaim

```
from ansys.geometry.core import launch_modeler_with_spaceclaim

modeler_discovery = launch_modeler_with_spaceclaim()
```

Geometry service

```python
from ansys.geometry.core import launch_modeler_with_geometry_service

modeler_discovery = launch_modeler_with_geometry_service()
```

For more information on the arguments accepted by the launcher methods, see their API documentation:

- launch_modeler_with_discovery
- launch_modeler_with_spaceclaim
- launch_modeler_with_geometry_service

Note: Because this is the first release of the Geometry service, you cannot yet define a product version or API version.

Go to Getting started

1.4.3 Launch a remote session

If a remote server is running Ansys 2023 R2 or later and is also running PIM (Product Instance Manager), you can use PIM to start a Discovery or SpaceClaim session that PyAnsys Geometry can connect to.

Warning: This option is only available for Ansys employees.

Only Ansys employees with credentials to the Artifact Repository Browser can download ZIP files for PIM.

Set up the client machine

1. To establish a connection to the existing session from your client machine, open Python and run these commands:

   ```python
   from ansys.discovery.core import launch_modeler_with_pimlight_and_discovery

   disco = launch_modeler_with_pimlight_and_discovery("241")
   ```

   The preceding commands launch a Discovery (version 24.1) session with the API server. You receive a `model` object back from Discovery that you then use as a PyAnsys Geometry client.

2. Start SpaceClaim or the Geometry service remotely using commands like these:

   ```python
   from ansys.discovery.core import launch_modeler_with_pimlight_and_spaceclaim

   sc = launch_modeler_with_pimlight_and_spaceclaim("version")

   from ansys.discovery.core import launch_modeler_with_pimlight_and_geometry_service

   geo = launch_modeler_with_pimlight_and_geometry_service("version")
   ```

Note: Performing all these operations remotely eliminates the need to worry about the starting endpoint or managing the session.

End the session

To end the session, run the corresponding command:

```
disco.close()
sc.close()
geo.close()
```

Go to Getting started

1.4.4 Use an existing session

If a session of Discovery, SpaceClaim, or the Geometry service is already running, PyAnsys Geometry can be used to connect to it.

Establish the connection

From Python, establish a connection to the existing client session by creating a `Modeler` object:

```
from ansys.geometry.core import Modeler

modeler = Modeler(host="localhost", port=5001)
```

If no error messages are received, your connection is established successfully. Note that your local port number might differ from the one shown in the preceding code.

Verify the connection

If you want to verify that the connection is successful, request the status of the client connection inside your `Modeler` object:

```
>>> modeler.client
Ansys Geometry Modeler Client (...)
Target:     localhost:5001
Connection: Healthy
```

Go to Getting started

1.4.5 Ansys version compatibility

The following table summarizes the compatibility matrix between the PyAnsys Geometry service and the Ansys product versions.

PyAnsys Geometry versions	Ansys Product versions	Geometry Service (dockerized)	Geometry Service (standalone)	Discovery	SpaceClaim
`0.2.X`	23R2				
`0.3.X`	23R2 (partially)				
`0.4.X`	24R1 onward				

Access to the documentation for the preceding versions is found at the Versions page.

Go to Getting started

1.4.6 Install package in development mode

This topic assumes that you want to install PyAnsys Geometry in developer mode so that you can modify the source and enhance it. You can install PyAnsys Geometry from PyPI or from the PyAnsys Geometry repository on GitHub.

Contents

Package dependencies

PyAnsys Geometry is supported on Python version 3.9 and later. As indicated in the Moving to require Python 3 statement, previous versions of Python are no longer supported.

PyAnsys Geometry dependencies are automatically checked when packages are installed. These projects are required dependencies for PyAnsys Geometry:

- ansys-api-geometry: Used for supplying gRPC code generated from Protobuf (PROTO) files
- NumPy: Used for data array access
- Pint: Used for measurement units
- PyVista: Used for interactive 3D plotting
- SciPy: Used for geometric transformations

PyPI

Before installing PyAnsys Geometry, to ensure that you have the latest version of pip, run this command:

```
python -m pip install -U pip
```

Then, to install PyAnsys Geometry, run this command:

```
python -m pip install ansys-geometry-core
```

GitHub

To install the latest release from the PyAnsys Geometry repository on GitHub, run these commands:

```
git clone https://github.com/ansys/pyansys-geometry
cd pyansys-geometry
pip install -e .
```

To verify your development installation, run this command:

```
tox
```

Install in offline mode

If you lack an internet connection on your installation machine (or you do not have access to the private Ansys PyPI packages repository), you should install PyAnsys Geometry by downloading the wheelhouse archive for your corresponding machine architecture from the repository's Releases page.

Each wheelhouse archive contains all the Python wheels necessary to install PyAnsys Geometry from scratch on Windows, Linux, and MacOS from Python 3.9 to 3.11. You can install this on an isolated system with a fresh Python installation or on a virtual environment.

For example, on Linux with Python 3.9, unzip the wheelhouse archive and install it with these commands:

```
unzip ansys-geometry-core-v0.4.dev0-wheelhouse-Linux-3.9.zip wheelhouse
pip install ansys-geometry-core -f wheelhouse --no-index --upgrade --ignore-installed
```

If you are on Windows with Python 3.9, unzip the wheelhouse archive to a wheelhouse directory and then install using the same `pip install` command as in the preceding example.

Consider installing using a virtual environment. For more information, see Creation of virtual environments in the Python documentation.

Verify your installation

Verify the *Modeler()* connection with this code:

```
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> print(modeler)

Ansys Geometry Modeler (0x205c5c17d90)

```

(continues on next page)

(continued from previous page)

```
Ansys Geometry Modeler Client (0x205c5c16e00)
Target:     localhost:652
Connection: Healthy
```

If you see a response from the server, you can start using PyAnsys Geometry as a service. For more information on PyAnsys Geometry usage, see *User guide*.

Go to Getting started

1.4.7 Frequently asked questions

What is PyAnsys?

PyAnsys is a set of open source Python libraries that allow you to interface with Ansys Electronics Desktop (AEDT), Ansys Mechanical, Ansys Parametric Design Language (APDL), Ansys Fluent, and other Ansys products.

You can use PyAnsys libraries within a Python environment of your choice in conjunction with external Python libraries.

What Ansys license do I need to run the Geometry service?

Note: This question is answered in https://github.com/ansys/pyansys-geometry/discussions/754.

The Ansys Geometry service is a headless service developed on top of the modeling libraries for Discovery and Space-Claim.

Both in its standalone and Docker versions, the Ansys Geometry service requires a **Discovery Modeling** license to run.

To run PyAnsys Geometry against other backends, such as Discovery or SpaceClaim, users must have an Ansys license that allows them to run these Ansys products.

The **Discovery Modeling** license is one of these licenses, but there are others, such as the Ansys Mechanical Enterprise license, that also allow users to run these Ansys products. However, the Geometry service is only compatible with the **Discovery Modeling** license.

Go to Getting started

CHAPTER

TWO

USER GUIDE

This section provides an overview of the PyAnsys Geometry library, explaining key concepts and approaches for primitives, sketches (2D basic shape elements), and model designs.

2.1 Primitives

The PyAnsys Geometry *primitives* subpackage consists of primitive representations of basic geometric objects, such as a point, vector, and matrix. To operate and manipulate physical quantities, this subpackage uses Pint, a third-party open source software that other PyAnsys libraries also use.

This table shows PyAnsys Geometry names and base values for the physical quantities:

Name	value
LENGTH_ACCURACY	1e-8
ANGLE_ACCURACY	1e-6
DEFAULT_UNITS.LENGTH	meter
DEFAULT_UNITS.ANGLE	radian

To define accuracy and measurements, you use these PyAnsys Geometry classes:

- *Accuracy()*
- *Measurements()*

2.1.1 Planes

The *Plane()* class provides primitive representation of a 2D plane in 3D space. It has an origin and a coordinate system. Sketched shapes are always defined relative to a plane. The default working plane is XY, which has `(0,0)` as its origin.

If you create a 2D object in the plane, PyAnsys Geometry converts it to the global coordinate system so that the 2D feature executes as expected:

```
origin = Point3D([42, 99, 13])
plane = Plane(origin, UnitVector3D([1, 0, 0]), UnitVector3D([0, 1, 0]))
```

2.2 Sketch

The PyAnsys Geometry *sketch* subpackage is used to build 2D basic shapes. Shapes consist of two fundamental constructs:

- **Edge**: A connection between two or more 2D points along a particular path. An edge represents an open shape such as an arc or line.
- **Face**: A set of edges that enclose a surface. A face represents a closed shape such as a circle or triangle.

To initialize a sketch, you first specify the *Plane()* class, which represents the plane in space from which other PyAnsys Geometry objects can be located.

This code shows how to initialize a sketch:

```
from ansys.geometry.core.sketch import Sketch

sketch = Sketch()
```

You then construct a sketch, which can be done using different approaches.

2.2.1 Functional-style API

A functional-style API is sometimes called a *fluent functional-style api* or *fluent API* in the developer community. However, to avoid confusion with the Ansys Fluent product, the PyAnsys Geometry documentation refrains from using the latter terms.

One of the key features of a functional-style API is that it keeps an active context based on the previously created edges to use as a reference starting point for additional objects.

The following code creates a sketch with its origin as a starting point. Subsequent calls create segments, which take as a starting point the last point of the previous edge.

```
sketch.segment_to_point(Point2D([3, 3]), "Segment2").segment_to_point(
    Point2D([3, 2]), "Segment3"
)
sketch.plot()
```

A functional-style API is also able to get a desired shape of the sketch object by taking advantage of user-defined labels:

```
sketch.get("<tag>")
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

2.2.2 Direct API

A direct API is sometimes called an *element-based approach* in the developer community.

This code shows how you can use a direct API to create multiple elements independently and combine them all together in a single plane:

```
sketch.triangle(
    Point2D([-10, 10]), Point2D([5, 6]), Point2D([-10, -10]), tag="triangle2"
)
sketch.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

For more information on sketch shapes, see the `Sketch()` subpackage.

2.3 Designer

The PyAnsys Geometry `designer` subpackage organizes geometry assemblies and synchronizes to a supporting Geometry service instance.

2.3.1 Define the model

The following code define the model by creating a sketch with a circle on the client. It then creates the model on the server.

```
# Create a sketch and draw a circle on the client
sketch = Sketch()
sketch.circle(Point3D([10, 10, 0], UNITS.mm), Quantity(10, UNITS.mm))

# Create your design on the server
design_name = "ExtrudeProfile"
design = modeler.create_design(design_name)
```

2.3.2 Add materials to model

This code adds the data structure and properties for individual materials:

```
density = Quantity(125, 1000 * UNITS.kg / (UNITS.m * UNITS.m * UNITS.m))
poisson_ratio = Quantity(0.33, UNITS.dimensionless)
tensile_strength = Quantity(45)
material = Material(
    "steel",
    density,
    [MaterialProperty(MaterialPropertyType.POISSON_RATIO, "myPoisson", poisson_ratio)],
)
material.add_property(MaterialPropertyType.TENSILE_STRENGTH, "myTensile", Quantity(45))
design.add_material(material)
```

2.3.3 Create bodies by extruding the sketch

Extruding a sketch projects all of the specified geometries onto the body. To create a solid body, this code extrudes the sketch profile by a given distance.

```
body = design.extrude_sketch("JustACircle", sketch, Quantity(10, UNITS.mm))
```

2.3.4 Create bodies by extruding the face

The following code shows how you can also extrude a face profile by a given distance to create a solid body. There are no modifications against the body containing the source face.

```
longer_body = design.extrude_face(
    "LongerCircleFace", body.faces[0], Quantity(20, UNITS.mm)
)
```

You can also translate and tessellate design bodies and project curves onto them. For more information, see these classes:

- *Body()*
- *Component()*

2.3.5 Download and save design

You can save your design to disk or download the design of the active Geometry server instance. The following code shows how to download and save the design.

```
file = "path/to/download"
design.download(file, as_stream=False)
```

For more information, see the *Design* submodule.

2.4 PyAnsys Geometry overview

PyAnsys Geometry is a Python wrapper for the Ansys Geometry service. Here are some of the key features of PyAnsys Geometry:

- Ability to use the library alongside other Python libraries
- A *functional-style* API for a clean and easy coding experience
- Built-in examples

2.5 Simple interactive example

This simple interactive example shows how to start an instance of the Geometry server and create a geometry model.

2.5.1 Start Geometry server instance

The *Modeler()* class within the `ansys-geometry-core` library creates an instance of the Geometry service. By default, the `Modeler` instance connects to `127.0.0.1` (`"localhost"`) on port `50051`. You can change this by modifying the `host` and `port` parameters of the `Modeler` object, but note that you must also modify your `docker run` command by changing the `<HOST-PORT>:50051` argument.

This code starts an instance of the Geometry service:

```
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
```

2.5.2 Create geometry model

Once an instance has started, you can create a geometry model by initializing the *Sketch* subpackage and using the *Primitives* subpackage.

```
from ansys.geometry.core.math import Plane, Point3D, Point2D
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.sketch import Sketch

# Define our sketch
origin = Point3D([0, 0, 10])
plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 1, 0])

# Create the sketch
sketch = Sketch(plane)
sketch.circle(Point2D([1, 1]), 30 * UNITS.m)
sketch.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

CHAPTER THREE

API REFERENCE

This section describes ansys-geometry-core endpoints, their capabilities, and how to interact with them programmatically.

3.1 The `ansys.geometry.core` library

3.1.1 Summary

Subpackages

connection	PyAnsys Geometry connection subpackage.
designer	PyAnsys Geometry designer subpackage.
materials	PyAnsys Geometry materials subpackage.
math	PyAnsys Geometry math subpackage.
misc	Provides the PyAnsys Geometry miscellaneous subpackage.
plotting	Provides the PyAnsys Geometry plotting subpackage.
primitives	PyAnsys Geometry primitives subpackage.
sketch	PyAnsys Geometry sketch subpackage.
tools	PyAnsys Geometry tools subpackage.

Submodules

errors	Provides PyAnsys Geometry-specific errors.
logger	Provides a general framework for logging in PyAnsys Geometry.
modeler	Provides for interacting with the Geometry service.
typing	Provides typing of values for PyAnsys Geometry.

Attributes

__version__	PyAnsys Geometry version.

Constants

USE_TRAME	Global constant for checking whether to use trame

The `connection` package

Summary

Submodules

backend	Module providing definitions for the backend types.
client	Module providing a wrapped abstraction of the gRPC PROTO API definition and stubs.
conversions	Module providing for conversions.
defaults	Module providing default connection parameters.
launcher	Module for connecting to instances of the Geometry service.
local_instance	Module for connecting to a local Docker container with the Geometry service.
product_instance	Module containing the `ProductInstance` class.
validate	Module to perform a connection validation check.

The `backend.py` module

Summary

Enums

BackendType	Provides an enum holding the available backend types.
ApiVersions	Provides an enum for all the compatibles API versions.

BackendType

class BackendType

Bases: `enum.Enum`

Provides an enum holding the available backend types.

Overview

Attributes

DISCOVERY
SPACECLAIM
WINDOWS_SERVICE
LINUX_SERVICE

Import detail

```
from ansys.geometry.core.connection.backend import BackendType
```

Attribute detail

`BackendType.`**`DISCOVERY`**` = 0`

`BackendType.`**`SPACECLAIM`**` = 1`

`BackendType.`**`WINDOWS_SERVICE`**` = 2`

`BackendType.`**`LINUX_SERVICE`**` = 3`

ApiVersions

`class ApiVersions`

Bases: `enum.Enum`

Provides an enum for all the compatibles API versions.

Overview

Attributes

V_21
V_22
V_231
V_232

Import detail

```
from ansys.geometry.core.connection.backend import ApiVersions
```

Attribute detail

`ApiVersions.V_21 = 21`

`ApiVersions.V_22 = 22`

`ApiVersions.V_231 = 231`

`ApiVersions.V_232 = 232`

Description

Module providing definitions for the backend types.

The `client.py` module

Summary

Classes

GrpcClient	Wraps the gRPC connection for the Geometry service.

Functions

wait_until_healthy	Wait until a channel is healthy before returning.

GrpcClient

class GrpcClient(*host: beartype.typing.Optional[str] = DEFAULT_HOST*, *port: beartype.typing.Union[str, int] = DEFAULT_PORT*, *channel: beartype.typing.Optional[grpc.Channel] = None*, *remote_instance: beartype.typing.Optional[ansys.platform.instancemanagement.Instance] = None*, *local_instance: beartype.typing.Optional[*ansys.geometry.core.connection.local_instance.LocalDockerInstance*] = None*, *product_instance: beartype.typing.Optional[*ansys.geometry.core.connection.product_instance.ProductInstance*] = None*, *timeout: beartype.typing.Optional[ansys.geometry.core.typing.Real] = 120*, *logging_level: beartype.typing.Optional[int] = logging.INFO*, *logging_file: beartype.typing.Optional[beartype.typing.Union[pathlib.Path, str]] = None*, *backend_type: beartype.typing.Optional[*ansys.geometry.core.connection.backend.BackendType*] = None*)

Wraps the gRPC connection for the Geometry service.

Overview

Methods

close	Close the channel.
target	Get the target of the channel.
get_name	Get the target name of the connection.

Properties

backend_type	Backend type.
channel	Client gRPC channel.
log	Specific instance logger.
is_closed	Flag indicating whether the client connection is closed.
healthy	Flag indicating whether the client channel is healthy.

Special methods

__repr__	Represent the client as a string.

Import detail

```
from ansys.geometry.core.connection.client import GrpcClient
```

Property detail

property GrpcClient.**backend_type**: ***BackendType***

Backend type.

Options are `Windows Service`, `Linux Service`, `Discovery`, and `SpaceClaim`.

Notes

This method might return `None` because determining the backend type is not straightforward.

property GrpcClient.**channel**: grpc.Channel

Client gRPC channel.

property GrpcClient.**log**: ***PyGeometryCustomAdapter***

Specific instance logger.

property GrpcClient.**is_closed**: bool

Flag indicating whether the client connection is closed.

property GrpcClient.**healthy**: bool

Flag indicating whether the client channel is healthy.

Method detail

`GrpcClient.__repr__()` → str

Represent the client as a string.

`GrpcClient.close()`

Close the channel.

Notes

If an instance of the Geometry service was started using PyPIM, this instance is deleted. Furthermore, if a local instance of the Geometry service was started, it is stopped.

`GrpcClient.target()` → str

Get the target of the channel.

`GrpcClient.get_name()` → str

Get the target name of the channel.

Description

Module providing a wrapped abstraction of the gRPC PROTO API definition and stubs.

Module detail

`client.wait_until_healthy`(*channel: grpc.Channel*, *timeout: float*)

Wait until a channel is healthy before returning.

Parameters

channel
[`Channel`] Channel that must be established and healthy.

timeout
[`float`] Timeout in seconds. An attempt is made every 100 milliseconds until the timeout is exceeded.

Raises

`TimeoutError`
Raised when the total elapsed time exceeds the value for the `timeout` parameter.

The `conversions.py` module

Summary

Functions

unit_vector_to_grpc_direction	Convert a `UnitVector3D` class to a unit vector Geometry service gRPC message.
frame_to_grpc_frame	Convert a `Frame` class to a frame Geometry service gRPC message.
plane_to_grpc_plane	Convert a `Plane` class to a plane Geometry service gRPC message.
sketch_shapes_to_grpc_geometri	Convert lists of `SketchEdge` and `SketchFace` to a `GRPCGeometries` message.
sketch_edges_to_grpc_geometrie	Convert a list of `SketchEdge` to a `GRPCGeometries` gRPC message.
sketch_arc_to_grpc_arc	Convert an `Arc` class to an arc Geometry service gRPC message.
sketch_ellipse_to_grpc_ellipse	Convert a `SketchEllipse` class to an ellipse Geometry service gRPC message.
sketch_circle_to_grpc_circle	Convert a `SketchCircle` class to a circle Geometry service gRPC message.
point3d_to_grpc_point	Convert a `Point3D` class to a point Geometry service gRPC message.
point2d_to_grpc_point	Convert a `Point2D` class to a point Geometry service gRPC message.
sketch_polygon_to_grpc_polygon	Convert a `Polygon` class to a polygon Geometry service gRPC message.
sketch_segment_to_grpc_line	Convert a `Segment` class to a line Geometry service gRPC message.
tess_to_pd	Convert an `ansys.api.geometry.Tessellation` to `pyvista.PolyData`.
grpc_matrix_to_matrix	Convert an `ansys.api.geometry.Matrix` to a `Matrix44`.
grpc_frame_to_frame	Convert an `ansys.api.geometry.Frame` gRPC message to a `Frame` class.

Description

Module providing for conversions.

Module detail

`conversions.`**`unit_vector_to_grpc_direction`**(*unit_vector:* ansys.geometry.core.math.vector.UnitVector3D) → ansys.api.geometry.v0.models_pb2.Direction

Convert a `UnitVector3D` class to a unit vector Geometry service gRPC message.

Parameters

unit_vector
[*UnitVector3D*] Source vector data.

Returns

`GRPCDirection`
Geometry service gRPC direction message.

`conversions.`**`frame_to_grpc_frame`**(*frame:* ansys.geometry.core.math.frame.Frame) → ansys.api.geometry.v0.models_pb2.Frame

Convert a `Frame` class to a frame Geometry service gRPC message.

Parameters

frame
[*Frame*] Source frame data.

Returns

`GRPCFrame`
Geometry service gRPC frame message. The unit for the frame origin is meters.

`conversions.`**`plane_to_grpc_plane`**`(`*plane:* ansys.geometry.core.math.plane.Plane`)` → ansys.api.geometry.v0.models_pb2.Plane

Convert a `Plane` class to a plane Geometry service gRPC message.

Parameters

plane
[*`Plane`*] Source plane data.

Returns

`GRPCPlane`
Geometry service gRPC plane message. The unit is meters.

`conversions.`**`sketch_shapes_to_grpc_geometries`**`(`*plane:* ansys.geometry.core.math.plane.Plane, *edges: beartype.typing.List[*ansys.geometry.core.sketch.edge.SketchEdge*]*, *faces: beartype.typing.List[*ansys.geometry.core.sketch.face.SketchFace*]*, *only_one_curve: beartype.typing.Optional[*bool*] = False*`)` → ansys.api.geometry.v0.models_pb2.Geometries

Convert lists of `SketchEdge` and `SketchFace` to a `GRPCGeometries` message.

Parameters

plane
[*`Plane`*] Plane for positioning the 2D sketches.

edges
[`List`[*`SketchEdge`*]] Source edge data.

faces
[`List`[*`SketchFace`*]] Source face data.

only_one_curve
[bool, default: `False`] Whether to project one curve of the whole set of geometries to enhance performance.

Returns

`GRPCGeometries`
Geometry service gRPC geometries message. The unit is meters.

`conversions.`**`sketch_edges_to_grpc_geometries`**`(`*edges: beartype.typing.List[*ansys.geometry.core.sketch.edge.SketchEdge*]*, *plane:* ansys.geometry.core.math.plane.Plane`)` → beartype.typing.Tuple[beartype.typing.List[ansys.api.geometry.v0.models_pb
beartype.typing.List[ansys.api.geometry.v0.models_pb2.Arc]]

Convert a list of `SketchEdge` to a `GRPCGeometries` gRPC message.

Parameters

edges
[`List`[*`SketchEdge`*]] Source edge data.

plane
[*`Plane`*] Plane for positioning the 2D sketches.

Returns

`Tuple[List[GRPCLine], List[GRPCArc]]`
Geometry service gRPC line and arc messages. The unit is meters.

`conversions.`**`sketch_arc_to_grpc_arc`**`(`*arc:* ansys.geometry.core.sketch.arc.Arc, *plane:* ansys.geometry.core.math.plane.Plane`)` → ansys.api.geometry.v0.models_pb2.Arc

Convert an `Arc` class to an arc Geometry service gRPC message.

Parameters

arc
[*`Arc`*] Source arc data.

plane
[*`Plane`*] Plane for positioning the arc within.

Returns

`GRPCArc`
Geometry service gRPC arc message. The unit is meters.

`conversions.`**`sketch_ellipse_to_grpc_ellipse`**`(`*ellipse:* ansys.geometry.core.sketch.ellipse.SketchEllipse, *plane:* ansys.geometry.core.math.plane.Plane`)` → ansys.api.geometry.v0.models_pb2.Ellipse

Convert a `SketchEllipse` class to an ellipse Geometry service gRPC message.

Parameters

ellipse
[*`SketchEllipse`*] Source ellipse data.

Returns

`GRPCEllipse`
Geometry service gRPC ellipse message. The unit is meters.

`conversions.`**`sketch_circle_to_grpc_circle`**`(`*circle:* ansys.geometry.core.sketch.circle.SketchCircle, *plane:* ansys.geometry.core.math.plane.Plane`)` → ansys.api.geometry.v0.models_pb2.Circle

Convert a `SketchCircle` class to a circle Geometry service gRPC message.

Parameters

circle
[*`SketchCircle`*] Source circle data.

plane
[*`Plane`*] Plane for positioning the circle.

Returns

`GRPCCircle`
Geometry service gRPC circle message. The unit is meters.

`conversions.`**`point3d_to_grpc_point`**`(`*point:* ansys.geometry.core.math.point.Point3D`)` → ansys.api.geometry.v0.models_pb2.Point

Convert a `Point3D` class to a point Geometry service gRPC message.

Parameters

point
[*`Point3D`*] Source point data.

Returns

`GRPCPoint`
Geometry service gRPC point message. The unit is meters.

`conversions.`**`point2d_to_grpc_point`**(*plane:* ansys.geometry.core.math.plane.Plane, *point2d:* ansys.geometry.core.math.point.Point2D) → ansys.api.geometry.v0.models_pb2.Point

Convert a `Point2D` class to a point Geometry service gRPC message.

Parameters

plane
[*Plane*] Plane for positioning the 2D point.

point
[*Point2D*] Source point data.

Returns

`GRPCPoint`
Geometry service gRPC point message. The unit is meters.

`conversions.`**`sketch_polygon_to_grpc_polygon`**(*polygon:* ansys.geometry.core.sketch.polygon.Polygon, *plane:* ansys.geometry.core.math.plane.Plane) → ansys.api.geometry.v0.models_pb2.Polygon

Convert a `Polygon` class to a polygon Geometry service gRPC message.

Parameters

polygon
[*Polygon*] Source polygon data.

Returns

`GRPCPolygon`
Geometry service gRPC polygon message. The unit is meters.

`conversions.`**`sketch_segment_to_grpc_line`**(*segment:* ansys.geometry.core.sketch.segment.SketchSegment, *plane:* ansys.geometry.core.math.plane.Plane) → ansys.api.geometry.v0.models_pb2.Line

Convert a `Segment` class to a line Geometry service gRPC message.

Parameters

segment
[*SketchSegment*] Source segment data.

Returns

`GRPCLine`
Geometry service gRPC line message. The unit is meters.

`conversions.`**`tess_to_pd`**(*tess: ansys.api.geometry.v0.models_pb2.Tessellation*) → pyvista.PolyData

Convert an `ansys.api.geometry.Tessellation` to `pyvista.PolyData`.

`conversions.`**`grpc_matrix_to_matrix`**(*m: ansys.api.geometry.v0.models_pb2.Matrix*) → *ansys.geometry.core.math.matrix.Matrix44*

Convert an `ansys.api.geometry.Matrix` to a `Matrix44`.

conversions.**grpc_frame_to_frame**(*frame: ansys.api.geometry.v0.models_pb2.Frame*) → *ansys.geometry.core.math.frame.Frame*

Convert an `ansys.api.geometry.Frame` gRPC message to a `Frame` class.

Parameters

GRPCFrame
Geometry service gRPC frame message. The unit for the frame origin is meters.

Returns

frame
[*Frame*] Resulting converted frame.

The `defaults.py` module

Summary

Constants

DEFAULT_HOST	Default for the HOST name.
DEFAULT_PORT	Default for the HOST port.
MAX_MESSAGE_LENGTH	Default for the gRPC maximum message length.
GEOMETRY_SERVICE_DOCKER_IMAGE	Default for the Geometry service Docker image location.
DEFAULT_PIM_CONFIG	Default for the PIM configuration when running PIM Light.

Description

Module providing default connection parameters.

Module detail

defaults.**DEFAULT_HOST**

Default for the HOST name.

By default, PyAnsys Geometry searches for the environment variable `ANSRV_GEO_HOST`, and if this variable does not exist, PyAnsys Geometry uses `127.0.0.1` as the host.

defaults.**DEFAULT_PORT**: int

Default for the HOST port.

By default, PyAnsys Geometry searches for the environment variable `ANSRV_GEO_PORT`, and if this variable does not exist, PyAnsys Geometry uses `50051` as the port.

defaults.**MAX_MESSAGE_LENGTH**

Default for the gRPC maximum message length.

By default, PyAnsys Geometry searches for the environment variable `PYGEOMETRY_MAX_MESSAGE_LENGTH`, and if this variable does not exist, it uses `256Mb` as the maximum message length.

defaults.**GEOMETRY_SERVICE_DOCKER_IMAGE = 'ghcr.io/ansys/geometry'**

Default for the Geometry service Docker image location.

Tag is dependent on what OS service is requested.

defaults.**DEFAULT_PIM_CONFIG**

Default for the PIM configuration when running PIM Light.

This parameter is only to be used when PIM Light is being run.

The `launcher.py` module

Summary

Functions

launch_modeler	Start the `Modeler` interface for PyAnsys Geometry.
launch_remote_modeler	Start the Geometry service remotely using the PIM API.
launch_local_modeler	Start the Geometry service locally using the `LocalDockerInstance` class.
launch_modeler_with_discovery_and_pimlig	Start Ansys Discovery remotely using the PIM API.
launch_modeler_with_geometry_service_and	Start the Geometry service remotely using the PIM API.
launch_modeler_with_spaceclaim_and_pimli	Start Ansys SpaceClaim remotely using the PIM API.
launch_modeler_with_geometry_service	Start the Geometry service locally using the `ProductInstance` class.
launch_modeler_with_discovery	Start Ansys Discovery locally using the `ProductInstance` class.
launch_modeler_with_spaceclaim	Start Ansys SpaceClaim locally using the `ProductInstance` class.

Description

Module for connecting to instances of the Geometry service.

Module detail

launcher.**launch_modeler**(***kwargs: beartype.typing.Optional[beartype.typing.Dict]*) → *ansys.geometry.core.modeler.Modeler*

Start the `Modeler` interface for PyAnsys Geometry.

Parameters

****kwargs**
[`dict`, default: `None`] Keyword arguments for the launching methods. For allowable keyword arguments, see the *launch_remote_modeler()* and *launch_local_modeler()* methods. Some of these keywords might be unused.

Returns

ansys.geometry.core.modeler.Modeler
Pythonic interface for geometry modeling.

Examples

Launch the Geometry service.

```
>>> from ansys.geometry.core import launch_modeler
>>> modeler = launch_modeler()
```

launcher.**launch_remote_modeler**(*version: beartype.typing.Optional[str] = None*, ***kwargs: beartype.typing.Optional[beartype.typing.Dict]*) → *ansys.geometry.core.modeler.Modeler*

Start the Geometry service remotely using the PIM API.

When calling this method, you must ensure that you are in an environment where PyPIM is configured. You can use the `pypim.is_configured` method to check if it is configured.

Parameters

version
[`str`, default: `None`] Version of the Geometry service to run in the three-digit format. For example, "232". If you do not specify the version, the server chooses the version.

****kwargs**
[`dict`, default: `None`] Keyword arguments for the launching methods. For allowable keyword arguments, see the *`launch_remote_modeler()`* and *`launch_local_modeler()`* methods. Some of these keywords might be unused.

Returns

`ansys.geometry.core.modeler.Modeler`
Instance of the Geometry service.

launcher.**launch_local_modeler**(*port: int = DEFAULT_PORT*, *connect_to_existing_service: bool = True*, *restart_if_existing_service: bool = False*, *name: beartype.typing.Optional[str] = None*, *image: beartype.typing.Optional[*ansys.geometry.core.connection.local_instance.GeometryContainers*] = None*, ***kwargs: beartype.typing.Optional[beartype.typing.Dict]*) → *ansys.geometry.core.modeler.Modeler*

Start the Geometry service locally using the `LocalDockerInstance` class.

When calling this method, a Geometry service (as a local Docker container) is started. By default, if a container with the Geometry service already exists at the given port, it connects to it. Otherwise, it tries to launch its own service.

Parameters

port
[`int`, `optional`] Localhost port to deploy the Geometry service on or the the `Modeler` interface to connect to (if it is already deployed). By default, the value is the one for the `DEFAULT_PORT` connection parameter.

connect_to_existing_service
[bool, default: `True`] Whether the `Modeler` interface should connect to a Geometry service already deployed at the specified port.

restart_if_existing_service
[bool, default: `False`] Whether the Geometry service (which is already running) should be restarted when attempting connection.

name
[`Optional`[`str`], default: `None`] Name of the Docker container to deploy. The default is `None`, in which case Docker assigns it a random name.

image
[`Optional`[*`GeometryContainers`*], default: `None`] The Geometry service Docker image to deploy. The default is `None`, in which case the `LocalDockerInstance` class identifies the OS of your Docker engine and deploys the latest version of the Geometry service for that OS.

****kwargs**
[`dict`, default: `None`] Keyword arguments for the launching methods. For allowable keyword arguments, see the *`launch_remote_modeler()`* and *`launch_local_modeler()`* methods. Some of these keywords might be unused.

Returns

`Modeler`
Instance of the Geometry service.

`launcher.`**`launch_modeler_with_discovery_and_pimlight`**`(`*version: beartype.typing.Optional[str] = None*`)` → *ansys.geometry.core.modeler.Modeler*

Start Ansys Discovery remotely using the PIM API.

When calling this method, you must ensure that you are in an environment where PyPIM is configured. You can use the `pypim.is_configured` method to check if it is configured.

Parameters

version
[`str`, default: `None`] Version of Discovery to run in the three-digit format. For example, “232”. If you do not specify the version, the server chooses the version.

Returns

`ansys.geometry.core.modeler.Modeler`
Instance of Modeler.

`launcher.`**`launch_modeler_with_geometry_service_and_pimlight`**`(`*version: beartype.typing.Optional[str] = None*`)` → *ansys.geometry.core.modeler.Modeler*

Start the Geometry service remotely using the PIM API.

When calling this method, you must ensure that you are in an environment where PyPIM is configured. You can use the `pypim.is_configured` method to check if it is configured.

Parameters

version
[`str`, default: `None`] Version of the Geometry service to run in the three-digit format. For example, “232”. If you do not specify the version, the server chooses the version.

Returns

`ansys.geometry.core.modeler.Modeler`
Instance of Modeler.

`launcher.`**`launch_modeler_with_spaceclaim_and_pimlight`**`(`*version: beartype.typing.Optional[str] = None*`)` → *ansys.geometry.core.modeler.Modeler*

Start Ansys SpaceClaim remotely using the PIM API.

When calling this method, you must ensure that you are in an environment where PyPIM is configured. You can use the `pypim.is_configured` method to check if it is configured.

Parameters

version
[`str`, default: `None`] Version of SpaceClaim to run in the three-digit format. For example, "232". If you do not specify the version, the server chooses the version.

Returns

`ansys.geometry.core.modeler.Modeler`
Instance of Modeler.

`launcher.`**`launch_modeler_with_geometry_service`**(*host: str = 'localhost', port: int = None, enable_trace: bool = False, log_level: int = 2, timeout: int = 60*) → *ansys.geometry.core.modeler.Modeler*

Start the Geometry service locally using the `ProductInstance` class.

When calling this method, a standalone Geometry service is started. By default, if an endpoint is specified (by defining *host* and *port* parameters) but the endpoint is not available, the startup will fail. Otherwise, it will try to launch its own service.

Parameters

host: str, optional
IP address at which the Geometry service will be deployed. By default, its value will be `localhost`.

port
[`int`, `optional`] Port at which the Geometry service will be deployed. By default, its value will be `None`.

enable_trace
[bool, `optional`] Boolean enabling the logs trace on the Geometry service console window. By default its value is `False`.

log_level
[`int`, `optional`] Backend's log level from 0 to 3:

- `0`: Chatterbox
- `1`: Debug
- `2`: Warning
- `3`: Error

The default is `2` (Warning).

timeout
[`int`, `optional`] Timeout for starting the backend startup process. The default is 60.

Returns

`Modeler`
Instance of the Geometry service.

Raises

`ConnectionError`
If the specified endpoint is already in use, a connection error will be raised.

`SystemError`
If there is not an Ansys product 23.2 version or later installed a SystemError will be raised.

Examples

Starting a geometry service with the default parameters and getting back a `Modeler` object:

```
>>> from ansys.geometry.core import launch_modeler_with_geometry_service
>>> modeler = launch_modeler_with_geometry_service()
```

Starting a geometry service, on address `10.171.22.44`, port `5001`, with chatty logs, traces enabled and a `300` seconds timeout:

```
>>> from ansys.geometry.core import launch_modeler_with_geometry_service
>>> modeler = launch_modeler_with_geometry_service(host="10.171.22.44",
    port=5001,
    log_level=0,
    enable_trace= True,
    timeout=300)
```

`launcher.`**`launch_modeler_with_discovery`**(*product_version: int = None*, *host: str = 'localhost'*, *port: int = None*, *log_level: int = 2*, *api_version: ansys.geometry.core.connection.backend.ApiVersions = ApiVersions.LATEST*, *timeout: int = 150*)

Start Ansys Discovery locally using the `ProductInstance` class.

Note: Support for Ansys Discovery is restricted to Ansys 24.1 onwards.

When calling this method, a standalone Discovery session is started. By default, if an endpoint is specified (by defining *host* and *port* parameters) but the endpoint is not available, the startup will fail. Otherwise, it will try to launch its own service.

Parameters

product_version: int, optional
The product version to be started. Goes from v23.2.1 to the latest. Default is `None`. If a specific product version is requested but not installed locally, a SystemError will be raised.

Ansys products versions and their corresponding int values:

- `241` : Ansys 24R1

host: str, optional
IP address at which the Discovery session will be deployed. By default, its value will be `localhost`.

port
[`int`, `optional`] Port at which the Geometry service will be deployed. By default, its value will be `None`.

log_level
[`int`, `optional`] Backend's log level from 0 to 3:

- `0`: Chatterbox
- `1`: Debug
- `2`: Warning
- `3`: Error

The default is `2` (Warning).

api_version: ApiVersions, optional
The backend's API version to be used at runtime. Goes from API v21 to the latest. Default is `ApiVersions.LATEST`.

timeout
[`int`, `optional`] Timeout for starting the backend startup process. The default is 150.

Returns

`Modeler`
Instance of the Geometry service.

Raises

`ConnectionError`
If the specified endpoint is already in use, a connection error will be raised.

SystemError:
If there is not an Ansys product 23.2 version or later installed or if a specific product's version is requested but not installed locally then a SystemError will be raised.

Examples

Starting an Ansys Discovery session with the default parameters and getting back a `Modeler` object:

```
>>> from ansys.geometry.core import launch_modeler_with_discovery
>>> modeler = launch_modeler_with_discovery()
```

Starting an Ansys Discovery V 23.2 session, on address `10.171.22.44`, port `5001`, with chatty logs, using API v231 and a `300` seconds timeout:

```
>>> from ansys.geometry.core import launch_modeler_with_discovery
>>> modeler = launch_modeler_with_discovery(product_version = 232,
    host="10.171.22.44",
    port=5001,
    log_level=0,
    api_version= 231,
    timeout=300)
```

`launcher.`**`launch_modeler_with_spaceclaim`**(*product_version: int = None*, *host: str = 'localhost'*, *port: int = None*, *log_level: int = 2*, *api_version: ansys.geometry.core.connection.backend.ApiVersions = ApiVersions.LATEST*, *timeout: int = 150*)

Start Ansys SpaceClaim locally using the `ProductInstance` class.

When calling this method, a standalone SpaceClaim session is started. By default, if an endpoint is specified (by defining *host* and *port* parameters) but the endpoint is not available, the startup will fail. Otherwise, it will try to launch its own service.

Parameters

product_version: int, optional
The product version to be started. Goes from v23.2.1 to the latest. Default is `None`. If a specific product version is requested but not installed locally, a SystemError will be raised.

Ansys products versions and their corresponding int values:

- `232` : Ansys 23R2 SP1

- 241 : Ansys 24R1

host: str, optional
IP address at which the SpaceClaim session will be deployed. By default, its value will be `localhost`.

port
[`int`, `optional`] Port at which the Geometry service will be deployed. By default, its value will be `None`.

log_level
[`int`, `optional`] Backend's log level from 0 to 3:

- `0`: Chatterbox
- `1`: Debug
- `2`: Warning
- `3`: Error

The default is `2` (Warning).

api_version: ApiVersions, optional
The backend's API version to be used at runtime. Goes from API v21 to the latest. Default is `ApiVersions.LATEST`.

timeout
[`int`, `optional`] Timeout for starting the backend startup process. The default is 150.

Returns

`Modeler`
Instance of the Geometry service.

Raises

`ConnectionError`
If the specified endpoint is already in use, a connection error will be raised.

`SystemError`
If there is not an Ansys product 23.2 version or later installed or if a specific product's version is requested but not installed locally then a SystemError will be raised.

Examples

Starting an Ansys SpaceClaim session with the default parameters and get back a `Modeler` object:

```
>>> from ansys.geometry.core import launch_modeler_with_spaceclaim
>>> modeler = launch_modeler_with_spaceclaim()
```

Starting an Ansys SpaceClaim V 23.2 session, on address `10.171.22.44`, port `5001`, with chatty logs, using API v231 and a `300` seconds timeout:

```
>>> from ansys.geometry.core import launch_modeler_with_spaceclaim
>>> modeler = launch_modeler_with_spaceclaim(product_version = 232,
    host="10.171.22.44",
    port=5001,
    log_level=0,
    api_version= 231,
    timeout=300)
```

The `local_instance.py` module

Summary

Classes

LocalDockerInstance	Instantiates a Geometry service as a local Docker container.

Enums

GeometryContainers	Provides an enum holding the available Geometry services.

Functions

get_geometry_container_type	Given a `LocalDockerInstance`, provide back the `GeometryContainers` value.

LocalDockerInstance

class LocalDockerInstance(*port: int = DEFAULT_PORT, connect_to_existing_service: bool = True, restart_if_existing_service: bool = False, name: beartype.typing.Optional[str] = None, image: beartype.typing.Optional[*GeometryContainers*] = None*)

Instantiates a Geometry service as a local Docker container.

Overview

Properties

container	Docker container object that hosts the deployed Geometry service.
existed_previously	Flag indicating whether the container previously existed.

Attributes

__DOCKER_CLIENT__	Docker client class variable. The default is `None`, in which case lazy

Static methods

docker_client	Get the initialized `__DOCKER_CLIENT__` object.
is_docker_installed	Check whether a local installation of Docker engine is available and running.

Import detail

```
from ansys.geometry.core.connection.local_instance import LocalDockerInstance
```

Property detail

property `LocalDockerInstance.`**`container`**: `docker.models.containers.Container`

Docker container object that hosts the deployed Geometry service.

property `LocalDockerInstance.`**`existed_previously`**: `bool`

Flag indicating whether the container previously existed.

Returns `False` if the Geometry service was effectively deployed by this class or `True` if it already existed.

Attribute detail

`LocalDockerInstance.`**`__DOCKER_CLIENT__`**: `docker.client.DockerClient`

Docker client class variable. The default is `None`, in which case lazy initialization is used.

Notes

`__DOCKER_CLIENT__` is a class variable, meaning that it is the same variable for all instances of this class.

Method detail

static `LocalDockerInstance.`**`docker_client`**`()` → docker.client.DockerClient

Get the initialized `__DOCKER_CLIENT__` object.

Returns

`DockerClient`
Initialized Docker client.

Notes

The `LocalDockerInstance` class performs a lazy initialization of the `__DOCKER_CLIENT__` class variable.

static `LocalDockerInstance.`**`is_docker_installed`**`()` → bool

Check whether a local installation of Docker engine is available and running.

Returns

bool
`True` if Docker engine is available and running, `False` otherwise.

GeometryContainers

class GeometryContainers

Bases: `enum.Enum`

Provides an enum holding the available Geometry services.

Overview

Attributes

WINDOWS_LATEST
LINUX_LATEST
WINDOWS_LATEST_UNSTABLE
LINUX_LATEST_UNSTABLE

Import detail

```
from ansys.geometry.core.connection.local_instance import GeometryContainers
```

Attribute detail

GeometryContainers.**WINDOWS_LATEST = (0, 'windows', 'windows-latest')**

GeometryContainers.**LINUX_LATEST = (1, 'linux', 'linux-latest')**

GeometryContainers.**WINDOWS_LATEST_UNSTABLE = (2, 'windows', 'windows-latest-unstable')**

GeometryContainers.**LINUX_LATEST_UNSTABLE = (3, 'linux', 'linux-latest-unstable')**

Description

Module for connecting to a local Docker container with the Geometry service.

Module detail

local_instance.**get_geometry_container_type**(*instance:* LocalDockerInstance) → beartype.typing.Union[*GeometryContainers*, None]

Given a `LocalDockerInstance`, provide back the `GeometryContainers` value.

Parameters

instance
[*LocalDockerInstance*] The LocalDockerInstance object.

Returns

Union[*GeometryContainers*, None]
The GeometryContainer value corresponding to the previous image or None if not match.

Notes

This method returns the first hit on the available tags.

The `product_instance.py` module

Summary

Classes

ProductInstance	`ProductInstance` class.

Functions

prepare_and_start_backend	Start the requested service locally using the `ProductInstance` class.
get_available_port	Return an available port to be used.

Constants

WINDOWS_GEOMETRY_SERVICE_FOLDER	Default Geometry Service's folder name into the unified installer.
DISCOVERY_FOLDER	Default Discovery's folder name into the unified installer.
SPACECLAIM_FOLDER	Default SpaceClaim's folder name into the unified installer.
ADDINS_SUBFOLDER	Default global Addins's folder name into the unified installer.
BACKEND_SUBFOLDER	Default backend's folder name into the `ADDINS_SUBFOLDER` folder.
MANIFEST_FILENAME	Default backend's addin filename.
GEOMETRY_SERVICE_EXE	The Windows Geometry Service's filename.
DISCOVERY_EXE	The Ansys Discovery's filename.
SPACECLAIM_EXE	The Ansys SpaceClaim's filename.
BACKEND_LOG_LEVEL_VARIABLE	The backend's log level environment variable for local start.
BACKEND_TRACE_VARIABLE	The backend's enable trace environment variable for local start.
BACKEND_HOST_VARIABLE	The backend's ip address environment variable for local start.
BACKEND_PORT_VARIABLE	The backend's port number environment variable for local start.
BACKEND_API_VERSION_VARIABLE	The backend's api version environment variable for local start.
BACKEND_SPACECLAIM_OPTIONS	The additional argument for local Ansys Discovery start.
BACKEND_ADDIN_MANIFEST_ARGUMENT	The argument to specify the backend's addin manifest file's path.

ProductInstance

class ProductInstance(*pid: int*)

`ProductInstance` class.

Overview

Methods

close	Close the process associated to the pid.

Import detail

```
from ansys.geometry.core.connection.product_instance import ProductInstance
```

Method detail

`ProductInstance.close()` → bool

Close the process associated to the pid.

Description

Module containing the `ProductInstance` class.

Module detail

`product_instance.prepare_and_start_backend`(*backend_type:* ansys.geometry.core.connection.backend.BackendType, *product_version: int = None*, *host: str = 'localhost'*, *port: int = None*, *enable_trace: bool = False*, *log_level: int = 2*, *api_version:* ansys.geometry.core.connection.backend.ApiVersions = *ApiVersions.LATEST*, *timeout: int = 150*) → *ansys.geometry.core.modeler.Modeler*

Start the requested service locally using the `ProductInstance` class.

When calling this method, a standalone service or product session is started. By default, if an endpoint is specified (by defining *host* and *port* parameters) but the endpoint is not available, the startup will fail. Otherwise, it will try to launch its own service.

Parameters

product_version: ``int``, optional
The product version to be started. Goes from v23.2.1 to the latest. Default is `None`. If a specific product version is requested but not installed locally, a SystemError will be raised.

host: str, optional
IP address at which the Geometry service will be deployed. By default, its value will be `localhost`.

port
[`int`, `optional`] Port at which the Geometry service will be deployed. By default, its value will be `None`.

enable_trace
[bool, `optional`] Boolean enabling the logs trace on the Geometry service console window. By default its value is `False`.

log_level
[`int`, `optional`]

Backend's log level from 0 to 3:
0: Chatterbox 1: Debug 2: Warning 3: Error

The default is 2 (Warning).

api_version: ``ApiVersions``, optional
The backend's API version to be used at runtime. Goes from API v21 to the latest. Default is `ApiVersions.LATEST`.

timeout
[`int`, `optional`] Timeout for starting the backend startup process. The default is 150.

Returns

Modeler
Instance of the Geometry service.

Raises

`ConnectionError`
If the specified endpoint is already in use, a connection error will be raised.

`SystemError`
If there is not an Ansys product 23.2 version or later installed or if a specific product's version is requested but not installed locally then a SystemError will be raised.

`product_instance.`**`get_available_port`**`()`

Return an available port to be used.

`product_instance.`**`WINDOWS_GEOMETRY_SERVICE_FOLDER = 'GeometryServices'`**

Default Geometry Service's folder name into the unified installer.

`product_instance.`**`DISCOVERY_FOLDER = 'Discovery'`**

Default Discovery's folder name into the unified installer.

`product_instance.`**`SPACECLAIM_FOLDER = 'scdm'`**

Default SpaceClaim's folder name into the unified installer.

`product_instance.`**`ADDINS_SUBFOLDER = 'Addins'`**

Default global Addins's folder name into the unified installer.

`product_instance.`**`BACKEND_SUBFOLDER = 'ApiServer'`**

Default backend's folder name into the `ADDINS_SUBFOLDER` folder.

`product_instance.`**`MANIFEST_FILENAME = 'Presentation.ApiServerAddIn.Manifest.xml'`**

Default backend's addin filename.

To be used only for local start of Ansys Discovery or Ansys SpaceClaim.

`product_instance.`**`GEOMETRY_SERVICE_EXE = 'Presentation.ApiServerDMS.exe'`**

The Windows Geometry Service's filename.

`product_instance.`**`DISCOVERY_EXE = 'Discovery.exe'`**

The Ansys Discovery's filename.

`product_instance.SPACECLAIM_EXE = 'SpaceClaim.exe'`

The Ansys SpaceClaim's filename.

`product_instance.BACKEND_LOG_LEVEL_VARIABLE = 'LOG_LEVEL'`

The backend's log level environment variable for local start.

`product_instance.BACKEND_TRACE_VARIABLE = 'ENABLE_TRACE'`

The backend's enable trace environment variable for local start.

`product_instance.BACKEND_HOST_VARIABLE = 'API_ADDRESS'`

The backend's ip address environment variable for local start.

`product_instance.BACKEND_PORT_VARIABLE = 'API_PORT'`

The backend's port number environment variable for local start.

`product_instance.BACKEND_API_VERSION_VARIABLE = 'API_VERSION'`

The backend's api version environment variable for local start.

To be used only with Ansys Discovery and Ansys SpaceClaim.

`product_instance.BACKEND_SPACECLAIM_OPTIONS = '--spaceclaim-options'`

The additional argument for local Ansys Discovery start.

To be used only with Ansys Discovery.

`product_instance.BACKEND_ADDIN_MANIFEST_ARGUMENT = '/ADDINMANIFESTFILE='`

The argument to specify the backend's addin manifest file's path.

To be used only with Ansys Discovery and Ansys SpaceClaim.

The `validate.py` module

Summary

Functions

validate	Create a client using the default settings and validate it.

Description

Module to perform a connection validation check.

The method in this module is only used for testing the default Docker service on GitHub and can safely be skipped within testing.

This command shows how this method is typically used:

```
python -c "from ansys.geometry.core.connection import validate; validate()"
```

Module detail

`validate.validate()`

Create a client using the default settings and validate it.

Description

PyAnsys Geometry connection subpackage.

The `designer` package

Summary

Submodules

beam	Provides for creating and managing a beam.
body	Provides for managing a body.
component	Provides for managing components.
coordinate_system	Provides for managing a user-defined coordinate system.
design	Provides for managing designs.
designpoint	Module for creating and managing design points.
edge	Module for managing an edge.
face	Module for managing a face.
part	Module providing fundamental data of an assembly.
selection	Module for creating a named selection.

The `beam.py` module

Summary

Classes

BeamProfile	Represents a single beam profile organized within the design assembly.
BeamCircularProfile	Represents a single circular beam profile organized within the design assembly.
Beam	Represents a simplified solid body with an assigned 2D cross-section.

BeamProfile

class BeamProfile(*id: str*, *name: str*)

Represents a single beam profile organized within the design assembly.

Overview

Properties

id	ID of the beam profile.
name	Name of the beam profile.

Import detail

```
from ansys.geometry.core.designer.beam import BeamProfile
```

Property detail

property `BeamProfile.`**`id`**`: str`

ID of the beam profile.

property `BeamProfile.`**`name`**`: str`

Name of the beam profile.

BeamCircularProfile

class BeamCircularProfile(*id: str*, *name: str*, *radius:* ansys.geometry.core.misc.measurements.Distance, *center:* ansys.geometry.core.math.point.Point3D, *direction_x:* ansys.geometry.core.math.vector.UnitVector3D, *direction_y:* ansys.geometry.core.math.vector.UnitVector3D)

Bases: *BeamProfile*

Represents a single circular beam profile organized within the design assembly.

Overview

Properties

radius	Radius of the circular beam profile.
center	Center of the circular beam profile.
direction_x	X-axis direction of the circular beam profile.
direction_y	Y-axis direction of the circular beam profile.

Special methods

`__repr__`	Represent the `BeamCircularProfile` as a string.

Import detail

```
from ansys.geometry.core.designer.beam import BeamCircularProfile
```

Property detail

property BeamCircularProfile.**radius**: *Distance*

Radius of the circular beam profile.

property BeamCircularProfile.**center**: *Point3D*

Center of the circular beam profile.

property BeamCircularProfile.**direction_x**: *UnitVector3D*

X-axis direction of the circular beam profile.

property BeamCircularProfile.**direction_y**: *UnitVector3D*

Y-axis direction of the circular beam profile.

Method detail

BeamCircularProfile.**__repr__**() → str

Represent the `BeamCircularProfile` as a string.

Beam

class Beam(*id: str*, *start:* ansys.geometry.core.math.point.Point3D, *end:* ansys.geometry.core.math.point.Point3D, *profile:* BeamProfile, *parent_component:* ansys.geometry.core.designer.component.Component)

Represents a simplified solid body with an assigned 2D cross-section.

Overview

Properties

`id`	Service-defined ID of the beam.
`start`	Start of the beam line segment.
`end`	End of the beam line segment.
`profile`	Beam profile of the beam line segment.
`parent_component`	Component node that the beam is under.
`is_alive`	Flag indicating whether the beam is still alive on the server side.

Special methods

__repr__	Represent the beam as a string.

Import detail

```
from ansys.geometry.core.designer.beam import Beam
```

Property detail

property Beam.**id**: str

Service-defined ID of the beam.

property Beam.**start**: *Point3D*

Start of the beam line segment.

property Beam.**end**: *Point3D*

End of the beam line segment.

property Beam.**profile**: *BeamProfile*

Beam profile of the beam line segment.

property Beam.**parent_component**: **beartype.typing.Union[*ansys.geometry.core.designer.component.Component*, None]**

Component node that the beam is under.

property Beam.**is_alive**: bool

Flag indicating whether the beam is still alive on the server side.

Method detail

Beam.**__repr__**() → str

Represent the beam as a string.

Description

Provides for creating and managing a beam.

The `body.py` module

Summary

Interfaces

IBody	Defines the common methods for a body, providing the abstract body interface.

Classes

MasterBody	Represents solids and surfaces organized within the design assembly.
Body	Represents solids and surfaces organized within the design assembly.

Enums

MidSurfaceOffsetType	Provides values for mid-surface offsets supported by the Geometry service.

IBody

class IBody

Bases: `abc.ABC`

Defines the common methods for a body, providing the abstract body interface.

Overview

Abstract methods

id	Get the ID of the body as a string.
name	Get the name of the body.
faces	Get a list of all faces within the body.
edges	Get a list of all edges within the body.
is_alive	Check if the body is still alive and has not been deleted.
is_surface	Check if the body is a planar body.
surface_thickness	Get the surface thickness of a surface body.
surface_offset	Get the surface offset type of a surface body.
volume	Calculate the volume of the body.
assign_material	Assign a material against the design in the active Geometry service instance.
add_midsurface_thickness	Add a mid-surface thickness to a surface body.
add_midsurface_offset	Add a mid-surface offset to a surface body.
imprint_curves	Imprint all specified geometries onto specified faces of the body.
project_curves	Project all specified geometries onto the body.
imprint_projected_curves	Project and imprint specified geometries onto the body.
translate	Translate the geometry body in the specified direction by a given distance.
copy	Create a copy of the body and place it under the specified parent component.
tessellate	Tessellate the body and return the geometry as triangles.
plot	Plot the body.

Methods

intersect	Intersect two bodies.
subtract	Subtract two bodies.
unite	Unite two bodies.

Import detail

```
from ansys.geometry.core.designer.body import IBody
```

Method detail

abstract IBody.**id**() → str

Get the ID of the body as a string.

abstract IBody.**name**() → str

Get the name of the body.

abstract IBody.**faces**() → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Get a list of all faces within the body.

Returns

List[*Face*]

abstract IBody.**edges**() → beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]

Get a list of all edges within the body.

Returns

List[*Edge*]

abstract IBody.**is_alive**() → bool

Check if the body is still alive and has not been deleted.

abstract IBody.**is_surface**() → bool

Check if the body is a planar body.

abstract IBody.**surface_thickness**() → beartype.typing.Union[pint.Quantity, None]

Get the surface thickness of a surface body.

Notes

This method is only for surface-type bodies that have been assigned a surface thickness.

abstract IBody.**surface_offset**() → beartype.typing.Union[*MidSurfaceOffsetType*, None]

Get the surface offset type of a surface body.

Notes

This method is only for surface-type bodies that have been assigned a surface offset.

`abstract IBody.volume()` → pint.Quantity

Calculate the volume of the body.

Notes

When dealing with a planar surface, a value of `0` is returned as a volume.

`abstract IBody.assign_material`(*material:* ansys.geometry.core.materials.material.Material) → None

Assign a material against the design in the active Geometry service instance.

Parameters

material
[*`Material`*] Source material data.

`abstract IBody.add_midsurface_thickness`(*thickness: pint.Quantity*) → None

Add a mid-surface thickness to a surface body.

Parameters

thickness
[`Quantity`] Thickness to assign.

Notes

Only surface bodies are eligible for mid-surface thickness assignment.

`abstract IBody.add_midsurface_offset`(*offset:* MidSurfaceOffsetType) → None

Add a mid-surface offset to a surface body.

Parameters

offset_type
[*`MidSurfaceOffsetType`*] Surface offset to assign.

Notes

Only surface bodies are eligible for mid-surface offset assignment.

`abstract IBody.imprint_curves`(*faces: beartype.typing.List[*ansys.geometry.core.designer.face.Face*]*, *sketch:* ansys.geometry.core.sketch.sketch.Sketch) → beartype.typing.Tuple[beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*], beartype.typing.List[*ansys.geometry.core.designer.face.Face*]]

Imprint all specified geometries onto specified faces of the body.

Parameters

faces: List[Face]
List of faces to imprint the curves of the sketch onto.

sketch: Sketch
All curves to imprint on the faces.

Returns

`Tuple[List[Edge], List[Face]]`
All impacted edges and faces from the imprint operation.

`abstract IBody.project_curves`(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *closest_face:* *bool*, *only_one_curve: beartype.typing.Optional[bool] = False*) → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Project all specified geometries onto the body.

Parameters

direction: UnitVector3D
Direction of the projection.

sketch: Sketch
All curves to project on the body.

closest_face: bool
Whether to target the closest face with the projection.

only_one_curve: bool, default: False
Whether to project only one curve of the entire sketch. When `True`, only one curve is projected.

Returns

`List[Face]`
All faces from the project curves operation.

Notes

The `only_one_curve` parameter allows you to optimize the server call because projecting curves is an expensive operation. This reduces the workload on the server side.

`abstract IBody.imprint_projected_curves`(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *closest_face:* *bool*, *only_one_curve: beartype.typing.Optional[bool] = False*) → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Project and imprint specified geometries onto the body.

This method combines the `project_curves()` and `imprint_curves()` method into one method. It has higher performance than calling them back-to-back when dealing with many curves. Because it is a specialized function, this method only returns the faces (and not the edges) from the imprint operation.

Parameters

direction: UnitVector3D
Direction of the projection.

sketch: Sketch
All curves to project on the body.

closest_face: bool
Whether to target the closest face with the projection.

only_one_curve: bool, default: False
Whether to project only one curve of the entire sketch. When `True`, only one curve is projected.

Returns

`List[`*`Face`*`]`
All imprinted faces from the operation.

Notes

The `only_one_curve` parameter allows you to optimize the server call because projecting curves is an expensive operation. This reduces the workload on the server side.

`abstract IBody.translate`(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*) → None

Translate the geometry body in the specified direction by a given distance.

Parameters

direction: UnitVector3D
Direction of the translation.

distance: Union[~pint.Quantity, Distance, Real]
Distance (magnitude) of the translation.

Returns

`None`

`abstract IBody.copy`(*parent:* ansys.geometry.core.designer.component.Component, *name: str = None*) → *Body*

Create a copy of the body and place it under the specified parent component.

Parameters

parent: Component
Parent component to place the new body under within the design assembly.

name: str
Name to give the new body.

Returns

Body
Copy of the body.

`abstract IBody.tessellate`(*merge: beartype.typing.Optional[bool] = False*) → beartype.typing.Union[pyvista.PolyData, pyvista.MultiBlock]

Tessellate the body and return the geometry as triangles.

Parameters

merge
[bool, default: `False`] Whether to merge the body into a single mesh. When `False` (default), the number of triangles are preserved and only the topology is merged. When `True`, the individual faces of the tessellation are merged.

Returns

`PolyData`, `MultiBlock`
Merged `pyvista.PolyData` if `merge=True` or a composite dataset.

Examples

Extrude a box centered at the origin to create a rectangular body and tessellate it:

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 0, 1])
>>> sketch = Sketch(plane)
>>> box = sketch.box(Point2D([2, 0]), 4, 4)
>>> design = modeler.create_design("my-design")
>>> my_comp = design.add_component("my-comp")
>>> body = my_comp.extrude_sketch("my-sketch", sketch, 1 * u.m)
>>> blocks = body.tessellate()
>>> blocks
>>> MultiBlock (0x7f94ec757460)
     N Blocks:  6
     X Bounds:  0.000, 4.000
     Y Bounds:  -1.000, 0.000
     Z Bounds:  -0.500, 4.500
```

Merge the body:

```
>>> mesh = body.tessellate(merge=True)
>>> mesh
PolyData (0x7f94ec75f3a0)
  N Cells:      12
  N Points:     24
  X Bounds:     0.000e+00, 4.000e+00
  Y Bounds:     -1.000e+00, 0.000e+00
  Z Bounds:     -5.000e-01, 4.500e+00
  N Arrays:     0
```

abstract IBody.**plot**(*merge: bool = False*, *screenshot: beartype.typing.Optional[str] = None*, *use_trame: beartype.typing.Optional[bool] = None*, ***plotting_options: beartype.typing.Optional[dict]*) → None

Plot the body.

Parameters

merge
[bool, default: `False`] Whether to merge the body into a single mesh. When `False` (default), the number of triangles are preserved and only the topology is merged. When `True`, the individual faces of the tessellation are merged.

screenshot
[`str`, default: `None`] Path for saving a screenshot of the image that is being represented.

use_trame
[bool, default: `None`] Whether to enable the use of trame. The default is `None`, in which case the `USE_TRAME` global setting is used.

****plotting_options**

[dict, default: None] Keyword arguments for plotting. For allowable keyword arguments, see the Plotter.add_mesh method.

Examples

Extrude a box centered at the origin to create rectangular body and plot it:

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 0, 1])
>>> sketch = Sketch(plane)
>>> box = sketch.box(Point2D([2, 0]), 4, 4)
>>> design = modeler.create_design("my-design")
>>> mycomp = design.add_component("my-comp")
>>> body = mycomp.extrude_sketch("my-sketch", sketch, 1 * u.m)
>>> body.plot()
```

Plot the body and color each face individually:

```
>>> body.plot(multi_colors=True)
```

IBody.**intersect**(*other:* Body) → None

Intersect two bodies.

Parameters

other
[*Body*] Body to intersect with.

Raises

ValueError
If the bodies do not intersect.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

IBody.**subtract**(*other:* Body) → None

Subtract two bodies.

Parameters

other
[*Body*] Body to subtract from the `self` parameter.

Raises

ValueError
If the subtraction results in an empty (complete) subtraction.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

`IBody.`**`unite`**`(`*other:* Body`)` → None

Unite two bodies.

Parameters

other
[*Body*] Body to unite with the `self` parameter.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

MasterBody

class MasterBody(*id: str*, *name: str*, *grpc_client:* ansys.geometry.core.connection.client.GrpcClient, *is_surface: bool = False*)

Bases: *IBody*

Represents solids and surfaces organized within the design assembly.

Overview

Abstract methods

imprint_curves	Imprint all specified geometries onto specified faces of the body.
project_curves	Project all specified geometries onto the body.
imprint_projected_curves	Project and imprint specified geometries onto the body.
intersect	Intersect two bodies.
subtract	Subtract two bodies.
unite	Unite two bodies.

Methods

reset_tessellation_cache	Decorate `MasterBody` methods that require a tessellation cache update.
assign_material	Assign a material against the design in the active Geometry service instance.
add_midsurface_thickness	Add a mid-surface thickness to a surface body.
add_midsurface_offset	Add a mid-surface offset to a surface body.
translate	Translate the geometry body in the specified direction by a given distance.
copy	Create a copy of the body and place it under the specified parent component.
tessellate	Tessellate the body and return the geometry as triangles.
plot	Plot the body.

Properties

id	Get the ID of the body as a string.
name	Get the name of the body.
is_surface	Check if the body is a planar body.
surface_thickness	Get the surface thickness of a surface body.
surface_offset	Get the surface offset type of a surface body.
faces	Get a list of all faces within the body.
edges	Get a list of all edges within the body.
is_alive	Check if the body is still alive and has not been deleted.
volume	Calculate the volume of the body.

Special methods

__repr__	Represent the master body as a string.

Import detail

```
from ansys.geometry.core.designer.body import MasterBody
```

Property detail

property MasterBody.**id: str**

Get the ID of the body as a string.

property MasterBody.**name: str**

Get the name of the body.

property MasterBody.**is_surface: bool**

Check if the body is a planar body.

property MasterBody.**surface_thickness: beartype.typing.Union[pint.Quantity, None]**

Get the surface thickness of a surface body.

Notes

This method is only for surface-type bodies that have been assigned a surface thickness.

property MasterBody.**surface_offset: beartype.typing.Union[*MidSurfaceOffsetType*, None]**

Get the surface offset type of a surface body.

Notes

This method is only for surface-type bodies that have been assigned a surface offset.

property MasterBody.**faces**: **beartype.typing.List[*ansys.geometry.core.designer.face.Face*]**

Get a list of all faces within the body.

Returns

`List[Face]`

property MasterBody.**edges**: **beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]**

Get a list of all edges within the body.

Returns

`List[Edge]`

property MasterBody.**is_alive**: bool

Check if the body is still alive and has not been deleted.

property MasterBody.**volume**: pint.Quantity

Calculate the volume of the body.

Notes

When dealing with a planar surface, a value of `0` is returned as a volume.

Method detail

MasterBody.**reset_tessellation_cache**()

Decorate `MasterBody` methods that require a tessellation cache update.

Parameters

func
[`method`] Method to call.

Returns

Any
Output of the method, if any.

MasterBody.**assign_material**(*material:* ansys.geometry.core.materials.material.Material) → None

Assign a material against the design in the active Geometry service instance.

Parameters

material
[*`Material`*] Source material data.

MasterBody.**add_midsurface_thickness**(*thickness: pint.Quantity*) → None

Add a mid-surface thickness to a surface body.

Parameters

thickness
[`Quantity`] Thickness to assign.

Notes

Only surface bodies are eligible for mid-surface thickness assignment.

`MasterBody.`**`add_midsurface_offset`**`(`*offset:* MidSurfaceOffsetType`)` → None

Add a mid-surface offset to a surface body.

Parameters

offset_type
[*`MidSurfaceOffsetType`*] Surface offset to assign.

Notes

Only surface bodies are eligible for mid-surface offset assignment.

`abstract` `MasterBody.`**`imprint_curves`**`(`*faces: beartype.typing.List[*ansys.geometry.core.designer.face.Face*]*, *sketch:* ansys.geometry.core.sketch.sketch.Sketch`)` → beartype.typing.Tuple[beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*], beartype.typing.List[*ansys.geometry.core.designer.face.Face*]]

Imprint all specified geometries onto specified faces of the body.

Parameters

faces: List[Face]
List of faces to imprint the curves of the sketch onto.

sketch: Sketch
All curves to imprint on the faces.

Returns

`Tuple[List[`*`Edge`*`], List[`*`Face`*`]]`
All impacted edges and faces from the imprint operation.

`abstract` `MasterBody.`**`project_curves`**`(`*direction:* ansys.geometry.core.math.vector.UnitVector3D, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *closest_face: bool*, *only_one_curve: beartype.typing.Optional[bool] = False*`)` → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Project all specified geometries onto the body.

Parameters

direction: UnitVector3D
Direction of the projection.

sketch: Sketch
All curves to project on the body.

closest_face: bool
Whether to target the closest face with the projection.

only_one_curve: bool, default: False
Whether to project only one curve of the entire sketch. When `True`, only one curve is projected.

Returns

`List[`*`Face`*`]`
All faces from the project curves operation.

Notes

The `only_one_curve` parameter allows you to optimize the server call because projecting curves is an expensive operation. This reduces the workload on the server side.

`abstract MasterBody.`**`imprint_projected_curves`**(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *closest_face:* bool, *only_one_curve: beartype.typing.Optional[bool] = False*) → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Project and imprint specified geometries onto the body.

This method combines the `project_curves()` and `imprint_curves()` method into one method. It has higher performance than calling them back-to-back when dealing with many curves. Because it is a specialized function, this method only returns the faces (and not the edges) from the imprint operation.

Parameters

direction: UnitVector3D
Direction of the projection.

sketch: Sketch
All curves to project on the body.

closest_face: bool
Whether to target the closest face with the projection.

only_one_curve: bool, default: False
Whether to project only one curve of the entire sketch. When `True`, only one curve is projected.

Returns

`List[`*`Face`*`]`
All imprinted faces from the operation.

Notes

The `only_one_curve` parameter allows you to optimize the server call because projecting curves is an expensive operation. This reduces the workload on the server side.

`MasterBody.`**`translate`**(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*) → None

Translate the geometry body in the specified direction by a given distance.

Parameters

direction: UnitVector3D
Direction of the translation.

distance: Union[~pint.Quantity, Distance, Real]
Distance (magnitude) of the translation.

Returns

`None`

`MasterBody.`**`copy`**(*parent:* ansys.geometry.core.designer.component.Component, *name: str = None*) → *Body*

Create a copy of the body and place it under the specified parent component.

Parameters

parent: Component
Parent component to place the new body under within the design assembly.

name: str
Name to give the new body.

Returns

Body
Copy of the body.

`MasterBody.`**`tessellate`**(*merge: beartype.typing.Optional[bool] = False*, *transform:* ansys.geometry.core.math.matrix.Matrix44 = *IDENTITY_MATRIX44*) → beartype.typing.Union[pyvista.PolyData, pyvista.MultiBlock]

Tessellate the body and return the geometry as triangles.

Parameters

merge
[bool, default: `False`] Whether to merge the body into a single mesh. When `False` (default), the number of triangles are preserved and only the topology is merged. When `True`, the individual faces of the tessellation are merged.

Returns

`PolyData`, **`MultiBlock`**
Merged `pyvista.PolyData` if `merge=True` or a composite dataset.

Examples

Extrude a box centered at the origin to create a rectangular body and tessellate it:

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 0, 1])
>>> sketch = Sketch(plane)
>>> box = sketch.box(Point2D([2, 0]), 4, 4)
>>> design = modeler.create_design("my-design")
>>> my_comp = design.add_component("my-comp")
>>> body = my_comp.extrude_sketch("my-sketch", sketch, 1 * u.m)
>>> blocks = body.tessellate()
>>> blocks
>>> MultiBlock (0x7f94ec757460)
     N Blocks:   6
     X Bounds:   0.000, 4.000
     Y Bounds:   -1.000, 0.000
     Z Bounds:   -0.500, 4.500
```

Merge the body:

```
>>> mesh = body.tessellate(merge=True)
>>> mesh
PolyData (0x7f94ec75f3a0)
  N Cells:      12
  N Points:     24
  X Bounds:     0.000e+00, 4.000e+00
  Y Bounds:     -1.000e+00, 0.000e+00
  Z Bounds:     -5.000e-01, 4.500e+00
  N Arrays:     0
```

`MasterBody.`**`plot`**(*merge: bool = False*, *screenshot: beartype.typing.Optional[str] = None*, *use_trame: beartype.typing.Optional[bool] = None*, ***plotting_options: beartype.typing.Optional[dict]*) → None

Plot the body.

Parameters

merge
[bool, default: `False`] Whether to merge the body into a single mesh. When `False` (default), the number of triangles are preserved and only the topology is merged. When `True`, the individual faces of the tessellation are merged.

screenshot
[`str`, default: `None`] Path for saving a screenshot of the image that is being represented.

use_trame
[bool, default: `None`] Whether to enable the use of trame. The default is `None`, in which case the `USE_TRAME` global setting is used.

****plotting_options**
[`dict`, default: `None`] Keyword arguments for plotting. For allowable keyword arguments, see the `Plotter.add_mesh` method.

Examples

Extrude a box centered at the origin to create rectangular body and plot it:

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 0, 1])
>>> sketch = Sketch(plane)
>>> box = sketch.box(Point2D([2, 0]), 4, 4)
>>> design = modeler.create_design("my-design")
>>> mycomp = design.add_component("my-comp")
>>> body = mycomp.extrude_sketch("my-sketch", sketch, 1 * u.m)
>>> body.plot()
```

Plot the body and color each face individually:

```
>>> body.plot(multi_colors=True)
```

abstract `MasterBody.`**`intersect`**(*other:* Body) → None

Intersect two bodies.

Parameters

other
[*Body*] Body to intersect with.

Raises

`ValueError`
If the bodies do not intersect.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

abstract `MasterBody.`**`subtract`**(*other:* Body) → None

Subtract two bodies.

Parameters

other
[*Body*] Body to subtract from the `self` parameter.

Raises

`ValueError`
If the subtraction results in an empty (complete) subtraction.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

abstract `MasterBody.`**`unite`**(*other:* Body) → None

Unite two bodies.

Parameters

other
[*Body*] Body to unite with the `self` parameter.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

`MasterBody.`**`__repr__`**() → str

Represent the master body as a string.

Body

class Body(*id*, *name*, *parent:* ansys.geometry.core.designer.component.Component, *template:* MasterBody)

Bases: *IBody*

Represents solids and surfaces organized within the design assembly.

Overview

Methods

reset_tessellation_cache	Decorate Body methods that require a tessellation cache update.
assign_material	Assign a material against the design in the active Geometry service instance.
add_midsurface_thickness	Add a mid-surface thickness to a surface body.
add_midsurface_offset	Add a mid-surface offset to a surface body.
imprint_curves	Imprint all specified geometries onto specified faces of the body.
project_curves	Project all specified geometries onto the body.
imprint_projected_curves	Project and imprint specified geometries onto the body.
translate	Translate the geometry body in the specified direction by a given distance.
copy	Create a copy of the body and place it under the specified parent component.
tessellate	Tessellate the body and return the geometry as triangles.
plot	Plot the body.
intersect	Intersect two bodies.
subtract	Subtract two bodies.
unite	Unite two bodies.

Properties

id	Get the ID of the body as a string.
name	Get the name of the body.
parent	
faces	Get a list of all faces within the body.
edges	Get a list of all edges within the body.
is_alive	Check if the body is still alive and has not been deleted.
is_surface	Check if the body is a planar body.
surface_thickness	Get the surface thickness of a surface body.
surface_offset	Get the surface offset type of a surface body.
volume	Calculate the volume of the body.

Special methods

__repr__	Represent the Body as a string.

Import detail

```
from ansys.geometry.core.designer.body import Body
```

Property detail

property Body.**id:** str

Get the ID of the body as a string.

property Body.**name:** str

Get the name of the body.

property Body.**parent:** *Component*

property Body.**faces:** **beartype.typing.List[*ansys.geometry.core.designer.face.Face*]**

Get a list of all faces within the body.

Returns

List[*Face*]

property Body.**edges:** **beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]**

Get a list of all edges within the body.

Returns

List[*Edge*]

property Body.**is_alive:** bool

Check if the body is still alive and has not been deleted.

property Body.**is_surface:** bool

Check if the body is a planar body.

property Body.**surface_thickness:** **beartype.typing.Union[pint.Quantity, None]**

Get the surface thickness of a surface body.

Notes

This method is only for surface-type bodies that have been assigned a surface thickness.

property Body.**surface_offset:** **beartype.typing.Union[*MidSurfaceOffsetType*, None]**

Get the surface offset type of a surface body.

Notes

This method is only for surface-type bodies that have been assigned a surface offset.

property Body.**volume**: pint.Quantity

Calculate the volume of the body.

Notes

When dealing with a planar surface, a value of `0` is returned as a volume.

Method detail

Body.**reset_tessellation_cache**()

Decorate `Body` methods that require a tessellation cache update.

Parameters

func
[`method`] Method to call.

Returns

`Any`
Output of the method, if any.

Body.**assign_material**(*material:* ansys.geometry.core.materials.material.Material) → None

Assign a material against the design in the active Geometry service instance.

Parameters

material
[*`Material`*] Source material data.

Body.**add_midsurface_thickness**(*thickness: pint.Quantity*) → None

Add a mid-surface thickness to a surface body.

Parameters

thickness
[`Quantity`] Thickness to assign.

Notes

Only surface bodies are eligible for mid-surface thickness assignment.

Body.**add_midsurface_offset**(*offset:* MidSurfaceOffsetType) → None

Add a mid-surface offset to a surface body.

Parameters

offset_type
[*`MidSurfaceOffsetType`*] Surface offset to assign.

Notes

Only surface bodies are eligible for mid-surface offset assignment.

`Body.`**`imprint_curves`**`(`*faces: beartype.typing.List[*ansys.geometry.core.designer.face.Face*]*, *sketch:* ansys.geometry.core.sketch.sketch.Sketch`)` → beartype.typing.Tuple[beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*], beartype.typing.List[*ansys.geometry.core.designer.face.Face*]]

Imprint all specified geometries onto specified faces of the body.

Parameters

faces: List[Face]
List of faces to imprint the curves of the sketch onto.

sketch: Sketch
All curves to imprint on the faces.

Returns

`Tuple`[`List`[*`Edge`*], `List`[*`Face`*]]
All impacted edges and faces from the imprint operation.

`Body.`**`project_curves`**`(`*direction:* ansys.geometry.core.math.vector.UnitVector3D, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *closest_face: bool*, *only_one_curve: beartype.typing.Optional[bool] = False*`)` → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Project all specified geometries onto the body.

Parameters

direction: UnitVector3D
Direction of the projection.

sketch: Sketch
All curves to project on the body.

closest_face: bool
Whether to target the closest face with the projection.

only_one_curve: bool, default: False
Whether to project only one curve of the entire sketch. When `True`, only one curve is projected.

Returns

`List`[*`Face`*]
All faces from the project curves operation.

Notes

The `only_one_curve` parameter allows you to optimize the server call because projecting curves is an expensive operation. This reduces the workload on the server side.

`Body.`**`imprint_projected_curves`**`(`*direction:* ansys.geometry.core.math.vector.UnitVector3D, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *closest_face: bool*, *only_one_curve: beartype.typing.Optional[bool] = False*`)` → beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Project and imprint specified geometries onto the body.

This method combines the `project_curves()` and `imprint_curves()` method into one method. It has higher performance than calling them back-to-back when dealing with many curves. Because it is a specialized function, this method only returns the faces (and not the edges) from the imprint operation.

Parameters

direction: UnitVector3D
Direction of the projection.

sketch: Sketch
All curves to project on the body.

closest_face: bool
Whether to target the closest face with the projection.

only_one_curve: bool, default: False
Whether to project only one curve of the entire sketch. When `True`, only one curve is projected.

Returns

`List[`***`Face`***`]`
All imprinted faces from the operation.

Notes

The `only_one_curve` parameter allows you to optimize the server call because projecting curves is an expensive operation. This reduces the workload on the server side.

`Body.`**`translate`**(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*) → None

Translate the geometry body in the specified direction by a given distance.

Parameters

direction: UnitVector3D
Direction of the translation.

distance: Union[~pint.Quantity, Distance, Real]
Distance (magnitude) of the translation.

Returns

`None`

`Body.`**`copy`**(*parent:* ansys.geometry.core.designer.component.Component, *name: str = None*) → *Body*

Create a copy of the body and place it under the specified parent component.

Parameters

parent: Component
Parent component to place the new body under within the design assembly.

name: str
Name to give the new body.

Returns

`Body`
Copy of the body.

`Body.tessellate`(*merge: beartype.typing.Optional[bool] = False*) → beartype.typing.Union[pyvista.PolyData, pyvista.MultiBlock]

Tessellate the body and return the geometry as triangles.

Parameters

merge
[bool, default: `False`] Whether to merge the body into a single mesh. When `False` (default), the number of triangles are preserved and only the topology is merged. When `True`, the individual faces of the tessellation are merged.

Returns

`PolyData`, `MultiBlock`
Merged `pyvista.PolyData` if `merge=True` or a composite dataset.

Examples

Extrude a box centered at the origin to create a rectangular body and tessellate it:

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 0, 1])
>>> sketch = Sketch(plane)
>>> box = sketch.box(Point2D([2, 0]), 4, 4)
>>> design = modeler.create_design("my-design")
>>> my_comp = design.add_component("my-comp")
>>> body = my_comp.extrude_sketch("my-sketch", sketch, 1 * u.m)
>>> blocks = body.tessellate()
>>> blocks
>>> MultiBlock (0x7f94ec757460)
     N Blocks:  6
     X Bounds:  0.000, 4.000
     Y Bounds:  -1.000, 0.000
     Z Bounds:  -0.500, 4.500
```

Merge the body:

```
>>> mesh = body.tessellate(merge=True)
>>> mesh
PolyData (0x7f94ec75f3a0)
  N Cells:      12
  N Points:     24
  X Bounds:     0.000e+00, 4.000e+00
  Y Bounds:     -1.000e+00, 0.000e+00
  Z Bounds:     -5.000e-01, 4.500e+00
  N Arrays:     0
```

`Body.plot`(*merge: bool = False, screenshot: beartype.typing.Optional[str] = None, use_trame: beartype.typing.Optional[bool] = None, **plotting_options: beartype.typing.Optional[dict]*) → None

Plot the body.

Parameters

merge
[bool, default: False] Whether to merge the body into a single mesh. When False (default), the number of triangles are preserved and only the topology is merged. When True, the individual faces of the tessellation are merged.

screenshot
[str, default: None] Path for saving a screenshot of the image that is being represented.

use_trame
[bool, default: None] Whether to enable the use of trame. The default is None, in which case the USE_TRAME global setting is used.

****plotting_options**
[dict, default: None] Keyword arguments for plotting. For allowable keyword arguments, see the Plotter.add_mesh method.

Examples

Extrude a box centered at the origin to create rectangular body and plot it:

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 0, 1])
>>> sketch = Sketch(plane)
>>> box = sketch.box(Point2D([2, 0]), 4, 4)
>>> design = modeler.create_design("my-design")
>>> mycomp = design.add_component("my-comp")
>>> body = mycomp.extrude_sketch("my-sketch", sketch, 1 * u.m)
>>> body.plot()
```

Plot the body and color each face individually:

```
>>> body.plot(multi_colors=True)
```

Body.**intersect**(*other:* Body) → None

Intersect two bodies.

Parameters

other
[*Body*] Body to intersect with.

Raises

ValueError
If the bodies do not intersect.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

`Body.`**`subtract`**`(`*other:* Body`)` → None

Subtract two bodies.

Parameters

other
[*Body*] Body to subtract from the `self` parameter.

Raises

`ValueError`
If the subtraction results in an empty (complete) subtraction.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

`Body.`**`unite`**`(`*other:* Body`)` → None

Unite two bodies.

Parameters

other
[*Body*] Body to unite with the `self` parameter.

Notes

The `self` parameter is directly modified with the result, and the `other` parameter is consumed. Thus, it is important to make copies if needed.

`Body.`**`__repr__`**`()` → str

Represent the `Body` as a string.

MidSurfaceOffsetType

`class MidSurfaceOffsetType`

Bases: `enum.Enum`

Provides values for mid-surface offsets supported by the Geometry service.

Overview

Attributes

MIDDLE
TOP
BOTTOM
VARIABLE
CUSTOM

Import detail

```
from ansys.geometry.core.designer.body import MidSurfaceOffsetType
```

Attribute detail

MidSurfaceOffsetType.**MIDDLE** = 0

MidSurfaceOffsetType.**TOP** = 1

MidSurfaceOffsetType.**BOTTOM** = 2

MidSurfaceOffsetType.**VARIABLE** = 3

MidSurfaceOffsetType.**CUSTOM** = 4

Description

Provides for managing a body.

The `component.py` module

Summary

Classes

Component	Provides for creating and managing a component.

Enums

SharedTopologyType	Enum for the component shared topologies available in the Geometry service.

Component

class Component(*name: str*, *parent_component: beartype.typing.Union[*Component, *None]*, *grpc_client:* ansys.geometry.core.connection.client.GrpcClient, *template: beartype.typing.Optional[*Component*] = None*, *preexisting_id: beartype.typing.Optional[str] = None*, *master_component: beartype.typing.Optional[*ansys.geometry.core.designer.part.MasterComponent*] = None*, *read_existing_comp: bool = False*)

Provides for creating and managing a component.

Overview

Methods

get_world_transform	Get the full transformation matrix of the component in world space.
modify_placement	Apply a translation and/or rotation to the existing placement matrix.
reset_placement	Reset a component's placement matrix to an identity matrix.
add_component	Add a new component under this component within the design assembly.
set_shared_topology	Set the shared topology to apply to the component.
extrude_sketch	Create a solid body by extruding the sketch profile up by a given distance.
extrude_face	Extrude the face profile by a given distance to create a solid body.
create_surface	Create a surface body with a sketch profile.
create_surface_from_face	Create a surface body based on a face.
create_coordinate_system	Create a coordinate system.
translate_bodies	Translate the geometry bodies in a specified direction by a given distance.
create_beams	Create beams under the component.
create_beam	Create a beam under the component.
delete_component	Delete a component (itself or its children).
delete_body	Delete a body belonging to this component (or its children).
add_design_point	Create a single design point.
add_design_points	Create a list of design points.
delete_beam	Delete an existing beam belonging to this component (or its children).
search_component	Search nested components recursively for a component.
search_body	Search bodies in the component and nested components recursively for a body.
search_beam	Search beams in the component and nested components recursively for a beam.
tessellate	Tessellate the component.
plot	Plot the component.

Properties

id	ID of the component.
name	Name of the component.
components	List of `Component` objects inside of the component.
bodies	List of `Body` objects inside of the component.
beams	List of `Beam` objects inside of the component.
design_points	List of `DesignPoint` objects inside of the component.
coordinate_systems	List of `CoordinateSystem` objects inside of the component.
parent_component	Parent of the component.
is_alive	Whether the component is still alive on the server side.
shared_topology	Shared topology type of the component (if any).

Special methods

__repr__	Represent the `Component` as a string.

Import detail

```
from ansys.geometry.core.designer.component import Component
```

Property detail

property Component.**id**: str

ID of the component.

property Component.**name**: str

Name of the component.

property Component.**components**: **beartype.typing.List[*Component*]**

List of `Component` objects inside of the component.

property Component.**bodies**: **beartype.typing.List[*ansys.geometry.core.designer.body.Body*]**

List of `Body` objects inside of the component.

property Component.**beams**: **beartype.typing.List[*ansys.geometry.core.designer.beam.Beam*]**

List of `Beam` objects inside of the component.

property Component.**design_points**: **beartype.typing.List[*ansys.geometry.core.designer.designpoint.DesignPoint*]**

List of `DesignPoint` objects inside of the component.

property Component.**coordinate_systems**: **beartype.typing.List[*ansys.geometry.core.designer.coordinate_system.CoordinateSystem*]**

List of `CoordinateSystem` objects inside of the component.

property Component.**parent_component**: **beartype.typing.Union[*Component*, None]**

Parent of the component.

property `Component.`**`is_alive`**`: bool`

Whether the component is still alive on the server side.

property `Component.`**`shared_topology`**`: beartype.typing.Union[`*`SharedTopologyType`*`, None]`

Shared topology type of the component (if any).

Notes

If no shared topology has been set, `None` is returned.

Method detail

`Component.`**`get_world_transform`**`()` → *ansys.geometry.core.math.matrix.Matrix44*

Get the full transformation matrix of the component in world space.

Returns

`Matrix44`
4x4 transformation matrix of the component in world space.

`Component.`**`modify_placement`**`(`*translation: beartype.typing.Optional[*ansys.geometry.core.math.vector.Vector3D*] = None*, *rotation_origin: beartype.typing.Optional[*ansys.geometry.core.math.point.Point3D*] = None*, *rotation_direction: beartype.typing.Optional[*ansys.geometry.core.math.vector.UnitVector3D*] = None*, *rotation_angle: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real] = 0*`)`

Apply a translation and/or rotation to the existing placement matrix.

Parameters

translation
[*`Vector3D`*, default: `None`] Vector that defines the desired translation to the component.

rotation_origin
[*`Point3D`*, default: `None`] Origin that defines the axis to rotate the component about.

rotation_direction
[*`UnitVector3D`*, default: `None`] Direction of the axis to rotate the component about.

rotation_angle
[`Union[Quantity,` *`Angle`*`, Real]`, default: 0] Angle to rotate the component around the axis.

Notes

To reset a component's placement to an identity matrix, see `reset_placement()` or call `modify_placement()` with no arguments.

`Component.`**`reset_placement`**`()`

Reset a component's placement matrix to an identity matrix.

See `modify_placement()`.

`Component.`**`add_component`**`(`*name:* *str*, *template: beartype.typing.Optional[*Component*] = None*`)` → *Component*

Add a new component under this component within the design assembly.

Parameters

name
[`str`] User-defined label for the new component.

template
[`Component`, default: `None`] Template to create this component from. This creates an instance component that shares a master with the template component.

Returns

`Component`
New component with no children in the design assembly.

`Component.`**`set_shared_topology`**`(`*share_type:* SharedTopologyType`)` → None

Set the shared topology to apply to the component.

Parameters

share_type
[`SharedTopologyType`] Shared topology type to assign to the component.

`Component.`**`extrude_sketch`**`(`*name:* *str*, *sketch:* ansys.geometry.core.sketch.sketch.Sketch, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*`)` → *ansys.geometry.core.designer.body.Body*

Create a solid body by extruding the sketch profile up by a given distance.

Parameters

name
[`str`] User-defined label for the new solid body.

sketch
[`Sketch`] Two-dimensional sketch source for the extrusion.

distance
[`Union[Quantity, Distance, Real]`] Distance to extrude the solid body.

Returns

`Body`
Extruded body from the given sketch.

Notes

The newly created body is placed under this component within the design assembly.

`Component.`**`extrude_face`**`(`*name: str*, *face:* ansys.geometry.core.designer.face.Face, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*]*`)` → *ansys.geometry.core.designer.body.Body*

Extrude the face profile by a given distance to create a solid body.

There are no modifications against the body containing the source face.

Parameters

name
[`str`] User-defined label for the new solid body.

face
[*`Face`*] Target face to use as the source for the new surface.

distance
[`Union`[`Quantity`, *`Distance`*]] Distance to extrude the solid body.

Returns

`Body`
Extruded solid body.

Notes

The source face can be anywhere within the design component hierarchy. Therefore, there is no validation requiring that the face is placed under the target component where the body is to be created.

`Component.`**`create_surface`**`(`*name: str*, *sketch:* ansys.geometry.core.sketch.sketch.Sketch`)` → *ansys.geometry.core.designer.body.Body*

Create a surface body with a sketch profile.

The newly created body is placed under this component within the design assembly.

Parameters

name
[`str`] User-defined label for the new surface body.

sketch
[*`Sketch`*] Two-dimensional sketch source for the surface definition.

Returns

`Body`
Body (as a planar surface) from the given sketch.

`Component.`**`create_surface_from_face`**`(`*name: str*, *face:* ansys.geometry.core.designer.face.Face`)` → *ansys.geometry.core.designer.body.Body*

Create a surface body based on a face.

Parameters

name
[`str`] User-defined label for the new surface body.

face
[`Face`] Target face to use as the source for the new surface.

Returns

`Body`
Surface body.

Notes

The source face can be anywhere within the design component hierarchy. Therefore, there is no validation requiring that the face is placed under the target component where the body is to be created.

`Component.`**`create_coordinate_system`**`(`*name: str*, *frame:* ansys.geometry.core.math.frame.Frame`)` → *ansys.geometry.core.designer.coordinate_system.CoordinateSystem*

Create a coordinate system.

The newly created coordinate system is place under this component within the design assembly.

Parameters

name
[`str`] User-defined label for the new coordinate system.

frame
[`Frame`] Frame defining the coordinate system bounds.

Returns

`CoordinateSystem`

`Component.`**`translate_bodies`**`(`*bodies: beartype.typing.List[*ansys.geometry.core.designer.body.Body*]*, *direction:* ansys.geometry.core.math.vector.UnitVector3D, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*`)` → None

Translate the geometry bodies in a specified direction by a given distance.

Parameters

bodies: List[Body]
List of bodies to translate by the same distance.

direction: UnitVector3D
Direction of the translation.

distance: Union[~pint.Quantity, Distance, Real]
Magnitude of the translation.

Returns

`None`

Notes

If the body does not belong to this component (or its children), it is not translated.

`Component.`**`create_beams`**(*segments: beartype.typing.List[beartype.typing.Tuple[*ansys.geometry.core.math.point.Point3D*,* ansys.geometry.core.math.point.Point3D*]]*, *profile:* ansys.geometry.core.designer.beam.BeamProfile) → beartype.typing.List[*ansys.geometry.core.designer.beam.Beam*]

Create beams under the component.

Parameters

segments
[`List`[`Tuple`[*`Point3D`*, *`Point3D`*]]] List of start and end pairs, each specifying a single line segment.

profile
[*`BeamProfile`*] Beam profile to use to create the beams.

Notes

The newly created beams synchronize to a design within a supporting Geometry service instance.

`Component.`**`create_beam`**(*start:* ansys.geometry.core.math.point.Point3D, *end:* ansys.geometry.core.math.point.Point3D, *profile:* ansys.geometry.core.designer.beam.BeamProfile) → *ansys.geometry.core.designer.beam.Beam*

Create a beam under the component.

The newly created beam synchronizes to a design within a supporting Geometry service instance.

Parameters

start
[*`Point3D`*] Starting point of the beam line segment.

end
[*`Point3D`*] Ending point of the beam line segment.

profile
[*`BeamProfile`*] Beam profile to use to create the beam.

`Component.`**`delete_component`**(*component: beartype.typing.Union[*Component*, str]*) → None

Delete a component (itself or its children).

Parameters

component
[`Union`[*`Component`*, `str`]] ID of the component or instance to delete.

Notes

If the component is not this component (or its children), it is not deleted.

`Component.`**`delete_body`**`(`*body: beartype.typing.Union[*ansys.geometry.core.designer.body.Body*, str]*`)` → None

Delete a body belonging to this component (or its children).

Parameters

body
[`Union`[*Body*, `str`]] ID of the body or instance to delete.

Notes

If the body does not belong to this component (or its children), it is not deleted.

`Component.`**`add_design_point`**`(`*name: str*, *point:* ansys.geometry.core.math.point.Point3D`)` → *ansys.geometry.core.designer.designpoint.DesignPoint*

Create a single design point.

Parameters

name
[`str`] User-defined label for the design points.

points
[*Point3D*] 3D point constituting the design point.

`Component.`**`add_design_points`**`(`*name: str*, *points: beartype.typing.List[*ansys.geometry.core.math.point.Point3D*]*`)` → beartype.typing.List[*ansys.geometry.core.designer.designpoint.DesignPoint*]

Create a list of design points.

Parameters

name
[`str`] User-defined label for the list of design points.

points
[`List`[*Point3D*]] List of the 3D points that constitute the list of design points.

`Component.`**`delete_beam`**`(`*beam: beartype.typing.Union[*ansys.geometry.core.designer.beam.Beam*, str]*`)` → None

Delete an existing beam belonging to this component (or its children).

Parameters

beam
[`Union`[*Beam*, `str`]] ID of the beam or instance to delete.

Notes

If the beam does not belong to this component (or its children), it is not deleted.

`Component.search_component`(*id: str*) → beartype.typing.Union[*Component*, None]

Search nested components recursively for a component.

Parameters

id
[`str`] ID of the component to search for.

Returns

`Component`
Component with the requested ID. If this ID is not found, `None` is returned.

`Component.search_body`(*id: str*) → beartype.typing.Union[*ansys.geometry.core.designer.body.Body*, None]

Search bodies in the component and nested components recursively for a body.

Parameters

id
[`str`] ID of the body to search for.

Returns

`Body`
Body with the requested ID. If the ID is not found, `None` is returned.

`Component.search_beam`(*id: str*) → beartype.typing.Union[*ansys.geometry.core.designer.beam.Beam*, None]

Search beams in the component and nested components recursively for a beam.

Parameters

id
[`str`] ID of the beam to search for.

Returns

`Union[Beam, None]`
Beam with the requested ID. If the ID is not found, `None` is returned.

`Component.tessellate`(*merge_component: bool = False*, *merge_bodies: bool = False*) → beartype.typing.Union[pyvista.PolyData, pyvista.MultiBlock]

Tessellate the component.

Parameters

merge_component
[bool, default: `False`] Whether to merge this component into a single dataset. When `True`, all the individual bodies are effectively combined into a single dataset without any hierarchy.

merge_bodies
[bool, default: `False`] Whether to merge each body into a single dataset. When `True`, all the faces of each individual body are effectively merged into a single dataset without separating faces.

Returns

`PolyData`, `MultiBlock`
Merged `pyvista.PolyData` if `merge_component=True` or a composite dataset.

Examples

Create two stacked bodies and return the tessellation as two merged bodies:

```
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core import Modeler
>>> from ansys.geometry.core.math import Point2D, Point3D, Plane
>>> from ansys.geometry.core.misc import UNITS
>>> from ansys.geometry.core.plotting import Plotter
>>> modeler = Modeler("10.54.0.72", "50051")
>>> sketch_1 = Sketch()
>>> box = sketch_1.box(
>>>    Point2D([10, 10], UNITS.m), Quantity(10, UNITS.m), Quantity(5, UNITS.m))
>>> sketch_1.circle(Point2D([0, 0], UNITS.m), Quantity(25, UNITS.m))
>>> design = modeler.create_design("MyDesign")
>>> comp = design.add_component("MyComponent")
>>> distance = Quantity(10, UNITS.m)
>>> body = comp.extrude_sketch("Body", sketch=sketch_1, distance=distance)
>>> sketch_2 = Sketch(Plane([0, 0, 10]))
>>> box = sketch_2.box(
>>>    Point2D([10, 10], UNITS.m), Quantity(10, UNITS.m), Quantity(5, UNITS.m))
>>> circle = sketch_2.circle(Point2D([0, 0], UNITS.m), Quantity(25, UNITS.m))
>>> body = comp.extrude_sketch("Body", sketch=sketch_2, distance=distance)
>>> dataset = comp.tessellate(merge_bodies=True)
>>> dataset
MultiBlock (0x7ff6bcb511e0)
  N Blocks:     2
  X Bounds:     -25.000, 25.000
  Y Bounds:     -24.991, 24.991
  Z Bounds:     0.000, 20.000
```

`Component.plot`(*merge_component: bool = False*, *merge_bodies: bool = False*, *screenshot: beartype.typing.Optional[str] = None*, *use_trame: beartype.typing.Optional[bool] = None*, ***plotting_options: beartype.typing.Optional[dict]*) → None

Plot the component.

Parameters

merge_component
[bool, default: `False`] Whether to merge the component into a single dataset. When `True`, all the individual bodies are effectively merged into a single dataset without any hierarchy.

merge_bodies
[bool, default: `False`] Whether to merge each body into a single dataset. When `True`, all the faces of each individual body are effectively merged into a single dataset without separating faces.

screenshot
[`str`, default: `None`] Path for saving a screenshot of the image being represented.

use_trame
[bool, default: `None`] Whether to enable the use of trame. The default is `None`, in which case the `USE_TRAME` global setting is used.

****plotting_options**
[`dict`, default: `None`] Keyword arguments for plotting. For allowable keyword arguments, see the

Examples

Create 25 small cylinders in a grid-like pattern on the XY plane and plot them. Make the cylinders look metallic by enabling physically-based rendering with pbr=True.

```
>>> from ansys.geometry.core.misc.units import UNITS as u
>>> from ansys.geometry.core.sketch import Sketch
>>> from ansys.geometry.core.math import Plane, Point2D, Point3D, UnitVector3D
>>> from ansys.geometry.core import Modeler
>>> import numpy as np
>>> modeler = Modeler()
>>> origin = Point3D([0, 0, 0])
>>> plane = Plane(origin, direction_x=[1, 0, 0], direction_y=[0, 1, 0])
>>> design = modeler.create_design("my-design")
>>> mycomp = design.add_component("my-comp")
>>> n = 5
>>> xx, yy = np.meshgrid(
...     np.linspace(-4, 4, n),
...     np.linspace(-4, 4, n),
... )
>>> for x, y in zip(xx.ravel(), yy.ravel()):
...     sketch = Sketch(plane)
...     sketch.circle(Point2D([x, y]), 0.2*u.m)
...     mycomp.extrude_sketch(f"body-{x}-{y}", sketch, 1 * u.m)
>>> mycomp
ansys.geometry.core.designer.Component 0x2203cc9ec50
    Name                 : my-comp
    Exists               : True
    Parent component     : my-design
    N Bodies             : 25
    N Components         : 0
    N Coordinate Systems : 0
>>> mycomp.plot(pbr=True, metallic=1.0)
```

Component.**__repr__**() → str

Represent the Component as a string.

SharedTopologyType

class SharedTopologyType

Bases: enum.Enum

Enum for the component shared topologies available in the Geometry service.

Overview

Attributes

SHARETYPE_NONE
SHARETYPE_SHARE
SHARETYPE_MERGE
SHARETYPE_GROUPS

Import detail

```
from ansys.geometry.core.designer.component import SharedTopologyType
```

Attribute detail

SharedTopologyType.**SHARETYPE_NONE** = 0

SharedTopologyType.**SHARETYPE_SHARE** = 1

SharedTopologyType.**SHARETYPE_MERGE** = 2

SharedTopologyType.**SHARETYPE_GROUPS** = 3

Description

Provides for managing components.

The `coordinate_system.py` module

Summary

Classes

CoordinateSystem	Represents a user-defined coordinate system within the design assembly.

CoordinateSystem

class CoordinateSystem(*name: str*, *frame:* ansys.geometry.core.math.frame.Frame, *parent_component:* ansys.geometry.core.designer.component.Component, *grpc_client:* ansys.geometry.core.connection.client.GrpcClient, *preexisting_id: beartype.typing.Optional[str] = None*)

Represents a user-defined coordinate system within the design assembly.

Overview

Properties

id	ID of the coordinate system.
name	Name of the coordinate system.
frame	Frame of the coordinate system.
parent_component	Parent component of the coordinate system.
is_alive	Flag indicating if coordinate system is still alive on the server side.

Special methods

__repr__	Represent the coordinate system as a string.

Import detail

```
from ansys.geometry.core.designer.coordinate_system import CoordinateSystem
```

Property detail

property CoordinateSystem.**id: str**

ID of the coordinate system.

property CoordinateSystem.**name: str**

Name of the coordinate system.

property CoordinateSystem.**frame: *Frame***

Frame of the coordinate system.

property CoordinateSystem.**parent_component: *Component***

Parent component of the coordinate system.

property CoordinateSystem.**is_alive: bool**

Flag indicating if coordinate system is still alive on the server side.

Method detail

CoordinateSystem.**__repr__**() → str

Represent the coordinate system as a string.

Description

Provides for managing a user-defined coordinate system.

The `design.py` module

Summary

Classes

Design	Provides for organizing geometry assemblies.

Enums

DesignFileFormat	Provides supported file formats that can be downloaded for designs.

Design

class Design(*name: str*, *grpc_client:* ansys.geometry.core.connection.client.GrpcClient, *read_existing_design: bool = False*)

Bases: *ansys.geometry.core.designer.component.Component*

Provides for organizing geometry assemblies.

Overview

Methods

add_material	Add a material to the design.
save	Save a design to disk on the active Geometry server instance.
download	Export and download the design from the active Geometry server instance.
create_named_selection	Create a named selection on the active Geometry server instance.
delete_named_selection	Delete a named selection on the active Geometry server instance.
delete_component	Delete a component (itself or its children).
set_shared_topology	Set the shared topology to apply to the component.
add_beam_circular_profile	Add a new beam circular profile under the design for the creating beams.
add_midsurface_thickness	Add a mid-surface thickness to a list of bodies.
add_midsurface_offset	Add a mid-surface offset type to a list of bodies.
delete_beam_profile	Remove a beam profile on the active geometry server instance.

Properties

design_id	The design's object unique id.
materials	List of materials available for the design.
named_selections	List of named selections available for the design.
beam_profiles	List of beam profile available for the design.

Special methods

__repr__	Represent the `Design` as a string.

Import detail

```
from ansys.geometry.core.designer.design import Design
```

Property detail

property Design.**design_id: str**

The design's object unique id.

property Design.**materials: beartype.typing.List[*ansys.geometry.core.materials.material.Material*]**

List of materials available for the design.

property Design.**named_selections: beartype.typing.List[*ansys.geometry.core.designer.selection.NamedSelection*]**

List of named selections available for the design.

property Design.**beam_profiles: beartype.typing.List[*ansys.geometry.core.designer.beam.BeamProfile*]**

List of beam profile available for the design.

Method detail

Design.**add_material**(*material:* ansys.geometry.core.materials.material.Material) → None

Add a material to the design.

Parameters

material
[`Material`] Material to add.

Design.**save**(*file_location: beartype.typing.Union[pathlib.Path, str]*) → None

Save a design to disk on the active Geometry server instance.

Parameters

file_location
[`Union[Path, str]`] Location on disk to save the file to.

`Design.download`(*file_location: beartype.typing.Union[pathlib.Path, str], format: beartype.typing.Optional[*DesignFileFormat*] = DesignFileFormat.SCDOCX*) → None

Export and download the design from the active Geometry server instance.

Parameters

file_location
[`Union`[`Path`, `str`]] Location on disk to save the file to.

format :DesignFileFormat, default: DesignFileFormat.SCDOCX
Format for the file to save to.

`Design.create_named_selection`(*name: str, bodies: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.body.Body*]] = None, faces: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.face.Face*]] = None, edges: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]] = None, beams: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.beam.Beam*]] = None, design_points: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.designpoint.Design
= None) → *ansys.geometry.core.designer.selection.NamedSelection*

Create a named selection on the active Geometry server instance.

Parameters

name
[`str`] User-defined name for the named selection.

bodies
[`List`[*`Body`*], default: `None`] All bodies to include in the named selection.

faces
[`List`[*`Face`*], default: `None`] All faces to include in the named selection.

edges
[`List`[*`Edge`*], default: `None`] All edges to include in the named selection.

beams
[`List`[*`Beam`*], default: `None`] All beams to include in the named selection.

design_points
[`List`[*`DesignPoint`*], default: `None`] All design points to include in the named selection.

Returns

`NamedSelection`
Newly created named selection that maintains references to all target entities.

`Design.delete_named_selection`(*named_selection: beartype.typing.Union[*ansys.geometry.core.designer.selection.NamedSelection*, str]*) → None

Delete a named selection on the active Geometry server instance.

Parameters

named_selection
[`Union`[*`NamedSelection`*, `str`]] Name of the named selection or instance.

`Design.`**`delete_component`**(*component: beartype.typing.Union[*ansys.geometry.core.designer.component.Component, *str]*) → None

Delete a component (itself or its children).

Parameters

id
[`Union`[*`Component`*, `str`]] Name of the component or instance to delete.

Raises

`ValueError`
The design itself cannot be deleted.

Notes

If the component is not this component (or its children), it is not deleted.

`Design.`**`set_shared_topology`**(*share_type:* ansys.geometry.core.designer.component.SharedTopologyType) → None

Set the shared topology to apply to the component.

Parameters

share_type
[*`SharedTopologyType`*] Shared topology type to assign.

Raises

`ValueError`
Shared topology does not apply to a design.

`Design.`**`add_beam_circular_profile`**(*name: str*, *radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*]*, *center: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*] = ZERO_POINT3D*, *direction_x: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X*, *direction_y: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Y*) → *ansys.geometry.core.designer.beam.BeamCircularProfile*

Add a new beam circular profile under the design for the creating beams.

Parameters

name
[`str`] User-defined label for the new beam circular profile.

radius
[`Real`] Radius of the beam circular profile.

center
[`Union`[`ndarray`, `RealSequence`, *`Point3D`*]] Center of the beam circular profile.

direction_x
[Union[ndarray, RealSequence, *UnitVector3D*, *Vector3D*]] X-plane direction.

direction_y
[Union[ndarray, RealSequence, *UnitVector3D*, *Vector3D*]] Y-plane direction.

Design.**add_midsurface_thickness**(*thickness: pint.Quantity*, *bodies: beartype.typing.List[*ansys.geometry.core.designer.body.Body*]*) → None

Add a mid-surface thickness to a list of bodies.

Parameters

thickness
[Quantity] Thickness to be assigned.

bodies
[List[*Body*]] All bodies to include in the mid-surface thickness assignment.

Notes

Only surface bodies will be eligible for mid-surface thickness assignment.

Design.**add_midsurface_offset**(*offset_type:* ansys.geometry.core.designer.body.MidSurfaceOffsetType, *bodies: beartype.typing.List[*ansys.geometry.core.designer.body.Body*]*) → None

Add a mid-surface offset type to a list of bodies.

Parameters

offset_type
[*MidSurfaceOffsetType*] Surface offset to be assigned.

bodies
[List[*Body*]] All bodies to include in the mid-surface offset assignment.

Notes

Only surface bodies will be eligible for mid-surface offset assignment.

Design.**delete_beam_profile**(*beam_profile: beartype.typing.Union[*ansys.geometry.core.designer.beam.BeamProfile, *str]*) → None

Remove a beam profile on the active geometry server instance.

Parameters

beam_profile
[Union[*BeamProfile*, str]] A beam profile name or instance that should be deleted.

Design.**__repr__**() → str

Represent the Design as a string.

DesignFileFormat

class DesignFileFormat

Bases: `enum.Enum`

Provides supported file formats that can be downloaded for designs.

Overview

Attributes

SCDOCX
PARASOLID_TEXT
PARASOLID_BIN
FMD
STEP
IGES
PMDB
INVALID

Import detail

```
from ansys.geometry.core.designer.design import DesignFileFormat
```

Attribute detail

DesignFileFormat.**SCDOCX = ('SCDOCX', None)**

DesignFileFormat.**PARASOLID_TEXT = ('PARASOLID_TEXT',)**

DesignFileFormat.**PARASOLID_BIN = ('PARASOLID_BIN',)**

DesignFileFormat.**FMD = ('FMD',)**

DesignFileFormat.**STEP = ('STEP',)**

DesignFileFormat.**IGES = ('IGES',)**

DesignFileFormat.**PMDB = ('PMDB',)**

DesignFileFormat.**INVALID = ('INVALID', None)**

Description

Provides for managing designs.

The `designpoint.py` module

Summary

Classes

DesignPoint	Provides for creating design points in components.

DesignPoint

class DesignPoint(*id: str*, *name: str*, *point:* ansys.geometry.core.math.point.Point3D, *parent_component:* ansys.geometry.core.designer.component.Component)

Provides for creating design points in components.

Overview

Properties

id	ID of the design point.
name	Name of the design point.
value	Value of the design point.
parent_component	Component node that the design point is under.

Special methods

__repr__	Represent the design points as a string.

Import detail

```python
from ansys.geometry.core.designer.designpoint import DesignPoint
```

Property detail

property DesignPoint.**id**: str

ID of the design point.

property DesignPoint.**name**: str

Name of the design point.

property DesignPoint.**value**: *Point3D*

Value of the design point.

property DesignPoint.**parent_component**: **beartype.typing.Union**[*ansys.geometry.core.designer.component.Component*, None]

Component node that the design point is under.

Method detail

DesignPoint.**__repr__**() → str

Represent the design points as a string.

Description

Module for creating and managing design points.

The `edge.py` module

Summary

Classes

Edge	Represents a single edge of a body within the design assembly.

Enums

CurveType	Provides values for the curve types supported by the Geometry service.

Edge

class Edge(*id*: str, *curve_type*: CurveType, *body*: ansys.geometry.core.designer.body.Body, *grpc_client*: ansys.geometry.core.connection.client.GrpcClient)

Represents a single edge of a body within the design assembly.

Overview

Properties

id	ID of the edge.
length	Calculated length of the edge.
curve_type	Curve type of the edge.
faces	Faces that contain the edge.
start_point	Edge start point.
end_point	Edge end point.

Import detail

```
from ansys.geometry.core.designer.edge import Edge
```

Property detail

property Edge.**id:** str

ID of the edge.

property Edge.**length:** pint.Quantity

Calculated length of the edge.

property Edge.**curve_type:** *CurveType*

Curve type of the edge.

property Edge.**faces:** beartype.typing.List[*ansys.geometry.core.designer.face.Face*]

Faces that contain the edge.

property Edge.**start_point:** *Point3D*

Edge start point.

property Edge.**end_point:** *Point3D*

Edge end point.

CurveType

class CurveType

Bases: enum.Enum

Provides values for the curve types supported by the Geometry service.

Overview

Attributes

CURVETYPE_UNKNOWN
CURVETYPE_LINE
CURVETYPE_CIRCLE
CURVETYPE_ELLIPSE
CURVETYPE_NURBS
CURVETYPE_PROCEDURAL

Import detail

```
from ansys.geometry.core.designer.edge import CurveType
```

Attribute detail

CurveType.**CURVETYPE_UNKNOWN = 0**

CurveType.**CURVETYPE_LINE = 1**

CurveType.**CURVETYPE_CIRCLE = 2**

CurveType.**CURVETYPE_ELLIPSE = 3**

CurveType.**CURVETYPE_NURBS = 4**

CurveType.**CURVETYPE_PROCEDURAL = 5**

Description

Module for managing an edge.

The `face.py` module

Summary

Classes

FaceLoop	Provides an internal class holding the face loops defined on the server side.
Face	Represents a single face of a body within the design assembly.

Enums

SurfaceType	Provides values for the surface types supported by the Geometry service.
FaceLoopType	Provides values for the face loop types supported by the Geometry service.

FaceLoop

class FaceLoop(*type:* FaceLoopType, *length:* *pint.Quantity*, *min_bbox:* ansys.geometry.core.math.point.Point3D, *max_bbox:* ansys.geometry.core.math.point.Point3D, *edges:* *beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]*)

Provides an internal class holding the face loops defined on the server side.

Overview

Properties

type	Type of the loop.
length	Length of the loop.
min_bbox	Minimum point of the bounding box containing the loop.
max_bbox	Maximum point of the bounding box containing the loop.
edges	Edges contained in the loop.

Import detail

```
from ansys.geometry.core.designer.face import FaceLoop
```

Property detail

property FaceLoop.**type**: *FaceLoopType*

Type of the loop.

property FaceLoop.**length**: pint.Quantity

Length of the loop.

property FaceLoop.**min_bbox**: *Point3D*

Minimum point of the bounding box containing the loop.

property FaceLoop.**max_bbox**: *Point3D*

Maximum point of the bounding box containing the loop.

property FaceLoop.**edges**: **beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]**

Edges contained in the loop.

Face

class Face(*id: str*, *surface_type:* SurfaceType, *body:* ansys.geometry.core.designer.body.Body, *grpc_client:* ansys.geometry.core.connection.client.GrpcClient)

Represents a single face of a body within the design assembly.

Overview

Methods

face_normal	Get the normal direction to the face evaluated at certain UV coordinates.
face_point	Get a point of the face evaluated at certain UV coordinates.

Properties

id	Face ID.
body	Body that the face belongs to.
area	Calculated area of the face.
surface_type	Surface type of the face.
edges	List of all edges of the face.
loops	List of all loops of the face.

Import detail

```
from ansys.geometry.core.designer.face import Face
```

Property detail

property Face.**id:** str

Face ID.

property Face.**body:** *Body*

Body that the face belongs to.

property Face.**area:** pint.Quantity

Calculated area of the face.

property Face.**surface_type:** *SurfaceType*

Surface type of the face.

property Face.**edges: beartype.typing.List[***ansys.geometry.core.designer.edge.Edge***]**

List of all edges of the face.

property Face.**loops: beartype.typing.List[***FaceLoop***]**

List of all loops of the face.

Method detail

`Face.`**`face_normal`**`(`*u: float = 0.5, v: float = 0.5*`)` → *ansys.geometry.core.math.vector.UnitVector3D*

Get the normal direction to the face evaluated at certain UV coordinates.

Parameters

u

[`float`, default: 0.5] First coordinate of the 2D representation of a surface in UV space. The default is `0.5`, which is the center of the surface.

v

[`float`, default: 0.5] Second coordinate of the 2D representation of a surface in UV space. The default is `0.5`, which is the center of the surface.

Returns

`UnitVector3D`

`UnitVector3D` object evaluated at the given U and V coordinates. This *`UnitVector3D`* object is perpendicular to the surface at the given UV coordinates.

Notes

To properly use this method, you must handle UV coordinates. Thus, you must know how these relate to the underlying Geometry service. It is an advanced method for Geometry experts only.

`Face.`**`face_point`**`(`*u: float = 0.5, v: float = 0.5*`)` → *ansys.geometry.core.math.point.Point3D*

Get a point of the face evaluated at certain UV coordinates.

Parameters

u

[`float`, default: 0.5] First coordinate of the 2D representation of a surface in UV space. The default is `0.5`, which is the center of the surface.

v

[`float`, default: 0.5] Second coordinate of the 2D representation of a surface in UV space. The default is `0.5`, which is the center of the surface.

Returns

`Point3D`

`Point3D` object evaluated at the given UV coordinates.

Notes

To properly use this method, you must handle UV coordinates. Thus, you must know how these relate to the underlying Geometry service. It is an advanced method for Geometry experts only.

SurfaceType

class SurfaceType

Bases: `enum.Enum`

Provides values for the surface types supported by the Geometry service.

Overview

Attributes

SURFACETYPE_UNKNOWN
SURFACETYPE_PLANE
SURFACETYPE_CYLINDER
SURFACETYPE_CONE
SURFACETYPE_TORUS
SURFACETYPE_SPHERE
SURFACETYPE_NURBS
SURFACETYPE_PROCEDURAL

Import detail

```
from ansys.geometry.core.designer.face import SurfaceType
```

Attribute detail

SurfaceType.**SURFACETYPE_UNKNOWN = 0**

SurfaceType.**SURFACETYPE_PLANE = 1**

SurfaceType.**SURFACETYPE_CYLINDER = 2**

SurfaceType.**SURFACETYPE_CONE = 3**

SurfaceType.**SURFACETYPE_TORUS = 4**

SurfaceType.**SURFACETYPE_SPHERE = 5**

SurfaceType.**SURFACETYPE_NURBS = 6**

SurfaceType.**SURFACETYPE_PROCEDURAL = 7**

FaceLoopType

class FaceLoopType

Bases: enum.Enum

Provides values for the face loop types supported by the Geometry service.

Overview

Attributes

INNER_LOOP
OUTER_LOOP

Import detail

```
from ansys.geometry.core.designer.face import FaceLoopType
```

Attribute detail

FaceLoopType.**INNER_LOOP = 'INNER'**

FaceLoopType.**OUTER_LOOP = 'OUTER'**

Description

Module for managing a face.

The part.py module

Summary

Classes

Part	Represents a part master.
MasterComponent	Represents a part occurrence.

Part

class Part(*id: str*, *name: str*, *components: beartype.typing.List[*MasterComponent*]*, *bodies: beartype.typing.List[*ansys.geometry.core.designer.body.MasterBody*]*)

Represents a part master.

Overview

Properties

id	ID of the part.
name	Name of the part.
components	`MasterComponent` children that the part contains.
bodies	`MasterBody` children that the part contains.

Special methods

__repr__	Represent the part as a string.

Import detail

```
from ansys.geometry.core.designer.part import Part
```

Property detail

property `Part.`**id**: `str`

ID of the part.

property `Part.`**name**: `str`

Name of the part.

property `Part.`**components**: **beartype.typing.List[*MasterComponent*]**

`MasterComponent` children that the part contains.

property `Part.`**bodies**: **beartype.typing.List[*ansys.geometry.core.designer.body.MasterBody*]**

`MasterBody` children that the part contains.

These are master bodies.

Method detail

`Part.__repr__()` → str

Represent the part as a string.

MasterComponent

class MasterComponent(*id: str*, *name: str*, *part:* Part, *transform:* ansys.geometry.core.math.matrix.Matrix44 = *IDENTITY_MATRIX44*)

Represents a part occurrence.

Overview

Properties

id	ID of the transformed part.
name	Name of the transformed part.
occurrences	List of all occurrences of the component.
part	Master part of the transformed part.
transform	4x4 transformation matrix from the master part.

Special methods

__repr__	Represent the master component as a string.

Import detail

```
from ansys.geometry.core.designer.part import MasterComponent
```

Property detail

property `MasterComponent.`**id**: str

ID of the transformed part.

property `MasterComponent.`**name**: str

Name of the transformed part.

property `MasterComponent.`**occurrences**: beartype.typing.List[*ansys.geometry.core.designer.component.Component*]

List of all occurrences of the component.

property `MasterComponent.`**part**: *Part*

Master part of the transformed part.

property `MasterComponent.`**`transform`**: ***`Matrix44`***

4x4 transformation matrix from the master part.

Method detail

`MasterComponent.`**`__repr__`**`()` → str

Represent the master component as a string.

Description

Module providing fundamental data of an assembly.

The `selection.py` module

Summary

Classes

`NamedSelection`	Represents a single named selection within the design assembly.

NamedSelection

class **`NamedSelection`**(*name:* str, *grpc_client:* ansys.geometry.core.connection.client.GrpcClient, *bodies: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.body.Body*]] = None*, *faces: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.face.Face*]] = None*, *edges: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.edge.Edge*]] = None*, *beams: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.beam.Beam*]] = None*, *design_points: beartype.typing.Optional[beartype.typing.List[*ansys.geometry.core.designer.designpoint.DesignPoint*]] = None*, *preexisting_id: beartype.typing.Optional[*str*] = None*)

Represents a single named selection within the design assembly.

Overview

Properties

`id`	ID of the named selection.
`name`	Name of the named selection.

Import detail

```
from ansys.geometry.core.designer.selection import NamedSelection
```

Property detail

property NamedSelection.**id**: str

ID of the named selection.

property NamedSelection.**name**: str

Name of the named selection.

Description

Module for creating a named selection.

Description

PyAnsys Geometry designer subpackage.

The `materials` package

Summary

Submodules

material	Provides the data structure for material and for adding a material property.
property	Provides the `MaterialProperty` class.

The `material.py` module

Summary

Classes

Material	Provides the data structure for a material.

Material

class Material(*name: str, density: pint.Quantity, additional_properties: beartype.typing.Optional[beartype.typing.Sequence[*ansys.geometry.core.materials.property.MaterialProperty*]] = None*)

Provides the data structure for a material.

Overview

Methods

add_property	Add a material property to the `Material` class.

Properties

properties	Dictionary of the material property type and material properties.
name	Material name.

Import detail

```
from ansys.geometry.core.materials.material import Material
```

Property detail

property Material.**properties: beartype.typing.Dict[*ansys.geometry.core.materials.property.MaterialPropertyType*, *ansys.geometry.core.materials.property.MaterialProperty*]**

Dictionary of the material property type and material properties.

property Material.**name: str**

Material name.

Method detail

Material.**add_property**(*type:* ansys.geometry.core.materials.property.MaterialPropertyType, *name: str*, *quantity: pint.Quantity*) → None

Add a material property to the `Material` class.

Parameters

type
[*MaterialPropertyType*] Material property type.

name: str
Material name.

quantity: ~pint.Quantity
Material value and unit.

Description

Provides the data structure for material and for adding a material property.

The `property.py` module

Summary

Classes

MaterialProperty	Provides the data structure for a material property.

Enums

MaterialPropertyType	Provides an enum holding the possible values for `MaterialProperty` objects.

MaterialProperty

class MaterialProperty(*type:* MaterialPropertyType, *name:* *str*, *quantity:* *pint.Quantity*)

Provides the data structure for a material property.

Overview

Properties

type	Material property ID.
name	Material property name.
quantity	Material property quantity and unit.

Import detail

```
from ansys.geometry.core.materials.property import MaterialProperty
```

Property detail

property `MaterialProperty`.**type**: ***MaterialPropertyType***

Material property ID.

property `MaterialProperty`.**name**: `str`

Material property name.

property `MaterialProperty`.**quantity**: `pint.Quantity`

Material property quantity and unit.

MaterialPropertyType

class MaterialPropertyType

Bases: `enum.Enum`

Provides an enum holding the possible values for `MaterialProperty` objects.

Overview

Methods

from_id	Return the `MaterialPropertyType` value from the service representation.

Attributes

DENSITY
ELASTIC_MODULUS
POISSON_RATIO
SHEAR_MODULUS
SPECIFIC_HEAT
TENSILE_STRENGTH
THERMAL_CONDUCTIVITY

Import detail

```
from ansys.geometry.core.materials.property import MaterialPropertyType
```

Attribute detail

`MaterialPropertyType.DENSITY = 'Density'`

`MaterialPropertyType.ELASTIC_MODULUS = 'ElasticModulus'`

`MaterialPropertyType.POISSON_RATIO = 'PoissonsRatio'`

`MaterialPropertyType.SHEAR_MODULUS = 'ShearModulus'`

`MaterialPropertyType.SPECIFIC_HEAT = 'SpecificHeat'`

`MaterialPropertyType.TENSILE_STRENGTH = 'TensileStrength'`

`MaterialPropertyType.THERMAL_CONDUCTIVITY = 'ThermalConductivity'`

Method detail

`MaterialPropertyType.from_id()` → *MaterialPropertyType*

Return the `MaterialPropertyType` value from the service representation.

Parameters

id
[`str`] Geometry Service string representation of a property type.

Returns

MaterialPropertyType
Common name for property type.

Description

Provides the `MaterialProperty` class.

Description

PyAnsys Geometry materials subpackage.

The `math` package

Summary

Submodules

bbox	Provides for managing a bounding box.
constants	Provides mathematical constants.
frame	Provides for managing a frame.
matrix	Provides matrix primitive representations.
plane	Provides primitive representation of a 2D plane in 3D space.
point	Provides geometry primitive representation for 2D and 3D points.
vector	Provides for creating and managing 2D and 3D vectors.

The `bbox.py` module

Summary

Classes

BoundingBox2D	Maintains the X and Y dimensions.

BoundingBox2D

class BoundingBox2D(*x_min: ansys.geometry.core.typing.Real = sys.float_info.max*, *x_max: ansys.geometry.core.typing.Real = sys.float_info.min*, *y_min: ansys.geometry.core.typing.Real = sys.float_info.max*, *y_max: ansys.geometry.core.typing.Real = sys.float_info.min*)

Maintains the X and Y dimensions.

Overview

Methods

add_point	Extend the ranges of the bounding box to include a point.
add_point_components	Extend the ranges of the bounding box to include the X and Y values.
add_points	Extend the ranges of the bounding box to include given points.
contains_point	Evaluate whether a provided point lies within the X and Y ranges of the bounds.
contains_point_components	Check if point components are within current X and Y ranges of the bounds.

Properties

x_min	Minimum value of X-dimensional bounds.
x_max	Maximum value of the X-dimensional bounds.
y_min	Minimum value of Y-dimensional bounds.
y_max	Maximum value of Y-dimensional bounds.

Special methods

__eq__	Equals operator for the `BoundingBox2D` class.
__ne__	Not equals operator for the `BoundingBox2D` class.

Import detail

```
from ansys.geometry.core.math.bbox import BoundingBox2D
```

Property detail

property BoundingBox2D.**x_min: Real**

Minimum value of X-dimensional bounds.

Returns

Real
Minimum value of the X-dimensional bounds.

property BoundingBox2D.**x_max: Real**

Maximum value of the X-dimensional bounds.

Returns

Real
Maximum value of the X-dimensional bounds.

property BoundingBox2D.**y_min: Real**

Minimum value of Y-dimensional bounds.

Returns

Real
Minimum value of Y-dimensional bounds.

property BoundingBox2D.**y_max: Real**

Maximum value of Y-dimensional bounds.

Returns

Real
Maximum value of Y-dimensional bounds.

Method detail

BoundingBox2D.**add_point**(*point:* ansys.geometry.core.math.point.Point2D) → None

Extend the ranges of the bounding box to include a point.

Parameters

point
[*Point2D*] Point to include within the bounds.

Notes

This method is only applicable if the point components are outside the current bounds.

`BoundingBox2D.`**`add_point_components`**`(`*x: ansys.geometry.core.typing.Real*, *y: ansys.geometry.core.typing.Real*`)` → None

Extend the ranges of the bounding box to include the X and Y values.

Parameters

x
[`Real`] Point X component to include within the bounds.

y
[`Real`] Point Y component to include within the bounds.

Notes

This method is only applicable if the point components are outside the current bounds.

`BoundingBox2D.`**`add_points`**`(`*points: beartype.typing.List[*ansys.geometry.core.math.point.Point2D*]*`)` → None

Extend the ranges of the bounding box to include given points.

Parameters

points
[`List`[*`Point2D`*]] List of points to include within the bounds.

`BoundingBox2D.`**`contains_point`**`(`*point:* ansys.geometry.core.math.point.Point2D`)` → bool

Evaluate whether a provided point lies within the X and Y ranges of the bounds.

Parameters

point
[*`Point2D`*] Point to compare against the bounds.

Returns

bool
`True` if the point is contained in the bounding box. Otherwise, `False`.

`BoundingBox2D.`**`contains_point_components`**`(`*x: ansys.geometry.core.typing.Real*, *y: ansys.geometry.core.typing.Real*`)` → bool

Check if point components are within current X and Y ranges of the bounds.

Parameters

x
[`Real`] Point X component to compare against the bounds.

y
[`Real`] Point Y component to compare against the bounds.

Returns

bool
`True` if the components are contained in the bounding box. Otherwise, `False`.

`BoundingBox2D.`**`__eq__`**`(`*other:* BoundingBox2D`)` → bool

Equals operator for the `BoundingBox2D` class.

`BoundingBox2D.__ne__`(*other:* BoundingBox2D) → bool

Not equals operator for the `BoundingBox2D` class.

Description

Provides for managing a bounding box.

The `constants.py` module

Summary

Constants

DEFAULT_POINT3D	Default value for a 3D point.
DEFAULT_POINT2D	Default value for a 2D point.
IDENTITY_MATRIX33	Identity for a `Matrix33` object.
IDENTITY_MATRIX44	Identity for a `Matrix44` object.
UNITVECTOR3D_X	Default 3D unit vector in the Cartesian traditional X direction.
UNITVECTOR3D_Y	Default 3D unit vector in the Cartesian traditional Y direction.
UNITVECTOR3D_Z	Default 3D unit vector in the Cartesian traditional Z direction.
UNITVECTOR2D_X	Default 2D unit vector in the Cartesian traditional X direction.
UNITVECTOR2D_Y	Default 2D unit vector in the Cartesian traditional Y direction.
ZERO_VECTOR3D	Zero-valued `Vector3D` object.
ZERO_VECTOR2D	Zero-valued `Vector2D` object.
ZERO_POINT3D	Zero-valued `Point3D` object.
ZERO_POINT2D	Zero-valued `Point2D` object.

Description

Provides mathematical constants.

Module detail

`constants.DEFAULT_POINT3D`

Default value for a 3D point.

`constants.DEFAULT_POINT2D`

Default value for a 2D point.

`constants.IDENTITY_MATRIX33`

Identity for a `Matrix33` object.

`constants.IDENTITY_MATRIX44`

Identity for a `Matrix44` object.

`constants.UNITVECTOR3D_X`

Default 3D unit vector in the Cartesian traditional X direction.

`constants.`**`UNITVECTOR3D_Y`**

Default 3D unit vector in the Cartesian traditional Y direction.

`constants.`**`UNITVECTOR3D_Z`**

Default 3D unit vector in the Cartesian traditional Z direction.

`constants.`**`UNITVECTOR2D_X`**

Default 2D unit vector in the Cartesian traditional X direction.

`constants.`**`UNITVECTOR2D_Y`**

Default 2D unit vector in the Cartesian traditional Y direction.

`constants.`**`ZERO_VECTOR3D`**

Zero-valued `Vector3D` object.

`constants.`**`ZERO_VECTOR2D`**

Zero-valued `Vector2D` object.

`constants.`**`ZERO_POINT3D`**

Zero-valued `Point3D` object.

`constants.`**`ZERO_POINT2D`**

Zero-valued `Point2D` object.

The `frame.py` module

Summary

Classes

`Frame`	Primitive representation of a frame (an origin and three fundamental directions).

Frame

`class Frame`(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*] = ZERO_POINT3D*, *direction_x: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X*, *direction_y: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Y*)

Primitive representation of a frame (an origin and three fundamental directions).

Overview

Methods

transform_point2d_local_to_global	Transform a 2D point to a global 3D point.

Properties

origin	Origin of the frame.
direction_x	X-axis direction of the frame.
direction_y	Y-axis direction of the frame.
direction_z	Z-axis direction of the frame.
global_to_local_rotation	Global to local space transformation matrix.
local_to_global_rotation	Local to global space transformation matrix.
transformation_matrix	Full 4x4 transformation matrix.

Special methods

__eq__	Equals operator for the `Frame` class.
__ne__	Not equals operator for the `Frame` class.

Import detail

```
from ansys.geometry.core.math.frame import Frame
```

Property detail

property Frame.**origin:** ***Point3D***

Origin of the frame.

property Frame.**direction_x:** ***UnitVector3D***

X-axis direction of the frame.

property Frame.**direction_y:** ***UnitVector3D***

Y-axis direction of the frame.

property Frame.**direction_z:** ***UnitVector3D***

Z-axis direction of the frame.

property Frame.**global_to_local_rotation:** ***Matrix33***

Global to local space transformation matrix.

Returns

Matrix33

3x3 matrix representing the transformation from global to local coordinate space, excluding origin translation.

property `Frame.`**`local_to_global_rotation:`** ***`Matrix33`***

Local to global space transformation matrix.

Returns

`Matrix33`
3x3 matrix representing the transformation from local to global coordinate space.

property `Frame.`**`transformation_matrix:`** ***`Matrix44`***

Full 4x4 transformation matrix.

Returns

`Matrix44`
4x4 matrix representing the transformation from global to local coordinate space.

Method detail

`Frame.`**`transform_point2d_local_to_global`**(*point:* ansys.geometry.core.math.point.Point2D) → *ansys.geometry.core.math.point.Point3D*

Transform a 2D point to a global 3D point.

This method transforms a local, plane-contained `Point2D` object in the global coordinate system, thus representing it as a `Point3D` object.

Parameters

point
[*`Point2D`*] `Point2D` local object to express in global coordinates.

Returns

`Point3D`
Global coordinates for the 3D point.

`Frame.`**`__eq__`**(*other:* Frame) → bool

Equals operator for the `Frame` class.

`Frame.`**`__ne__`**(*other:* Frame) → bool

Not equals operator for the `Frame` class.

Description

Provides for managing a frame.

The `matrix.py` module

Summary

Classes

`Matrix`	Provides matrix primitive representation.
`Matrix33`	Provides 3x3 matrix primitive representation.
`Matrix44`	Provides 4x4 matrix primitive representation.

Constants

DEFAULT_MATRIX33	Default value of the 3x3 identity matrix for the `Matrix33` class.
DEFAULT_MATRIX44	Default value of the 4x4 identity matrix for the `Matrix44` class.

Matrix

class Matrix(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: `numpy.ndarray`

Provides matrix primitive representation.

Overview

Methods

determinant	Get the determinant of the matrix.
inverse	Provide the inverse of the matrix.

Special methods

__mul__	Get the multiplication of the matrix.
__eq__	Equals operator for the `Matrix` class.
__ne__	Not equals operator for the `Matrix` class.

Import detail

```
from ansys.geometry.core.math.matrix import Matrix
```

Method detail

`Matrix.`**`determinant`**`()` → ansys.geometry.core.typing.Real

Get the determinant of the matrix.

`Matrix.`**`inverse`**`()` → *Matrix*

Provide the inverse of the matrix.

`Matrix.`**`__mul__`**(*other: beartype.typing.Union[*Matrix, *numpy.ndarray]*) → *Matrix*

Get the multiplication of the matrix.

`Matrix.`**`__eq__`**(*other:* Matrix) → bool

Equals operator for the `Matrix` class.

`Matrix.`**`__ne__`**(*other:* Matrix) → bool

Not equals operator for the `Matrix` class.

Matrix33

class Matrix33(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: *Matrix*

Provides 3x3 matrix primitive representation.

Import detail

```
from ansys.geometry.core.math.matrix import Matrix33
```

Matrix44

class Matrix44(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: *Matrix*

Provides 4x4 matrix primitive representation.

Import detail

```
from ansys.geometry.core.math.matrix import Matrix44
```

Description

Provides matrix primitive representations.

Module detail

matrix.**DEFAULT_MATRIX33**

Default value of the 3x3 identity matrix for the `Matrix33` class.

matrix.**DEFAULT_MATRIX44**

Default value of the 4x4 identity matrix for the `Matrix44` class.

The `plane.py` module

Summary

Classes

Plane	Provides primitive representation of a 2D plane in 3D space.

Plane

class Plane(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*] = ZERO_POINT3D, direction_x: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X, direction_y: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Y*)

Bases: `ansys.geometry.core.math.frame.Frame`

Provides primitive representation of a 2D plane in 3D space.

Overview

Methods

`is_point_contained`	Check if a 3D point is contained in the plane.

Special methods

`__eq__`	Equals operator for the `Plane` class.
`__ne__`	Not equals operator for the `Plane` class.

Import detail

```python
from ansys.geometry.core.math.plane import Plane
```

Method detail

`Plane.`**`is_point_contained`**(*point:* ansys.geometry.core.math.point.Point3D) → bool

Check if a 3D point is contained in the plane.

Parameters

point
[`Point3D`] `Point3D` class to check.

Returns

bool
`True` if the 3D point is contained in the plane, `False` otherwise.

`Plane.`**`__eq__`**(*other:* Plane) → bool

Equals operator for the `Plane` class.

`Plane.`**`__ne__`**(*other:* Plane) → bool

Not equals operator for the `Plane` class.

Description

Provides primitive representation of a 2D plane in 3D space.

The `point.py` module

Summary

Classes

Point2D	Provides geometry primitive representation for a 2D point.
Point3D	Provides geometry primitive representation for a 3D point.

Constants

DEFAULT_POINT2D_VALUES	Default values for a 2D point.
DEFAULT_POINT3D_VALUES	Default values for a 3D point.
BASE_UNIT_LENGTH	Default value for the length of the base unit.

Point2D

class Point2D(*input: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence] = DEFAULT_POINT2D_VALUES*, *unit: beartype.typing.Optional[pint.Unit] = None*)

Bases: `numpy.ndarray`, *ansys.geometry.core.misc.units.PhysicalQuantity*

Provides geometry primitive representation for a 2D point.

Overview

Methods

unit	Get the unit of the object.
base_unit	Get the base unit of the object.

Properties

x	X plane component value.
y	Y plane component value.

Special methods

`__eq__`	Equals operator for the `Point2D` class.
`__ne__`	Not equals operator for the `Point2D` class.
`__add__`	Add operation for the `Point2D` class.
`__sub__`	Subtraction operation for the `Point2D` class.

Import detail

```
from ansys.geometry.core.math.point import Point2D
```

Property detail

property `Point2D.`**`x`**: `pint.Quantity`

X plane component value.

property `Point2D.`**`y`**: `pint.Quantity`

Y plane component value.

Method detail

`Point2D.`**`__eq__`**(*other:* Point2D) → bool

Equals operator for the `Point2D` class.

`Point2D.`**`__ne__`**(*other:* Point2D) → bool

Not equals operator for the `Point2D` class.

`Point2D.`**`__add__`**(*other: beartype.typing.Union[*Point2D, ansys.geometry.core.math.vector.Vector2D*]*) → *Point2D*

Add operation for the `Point2D` class.

`Point2D.`**`__sub__`**(*other:* Point2D) → *Point2D*

Subtraction operation for the `Point2D` class.

`Point2D.`**`unit`**() → pint.Unit

Get the unit of the object.

`Point2D.`**`base_unit`**() → pint.Unit

Get the base unit of the object.

Point3D

class Point3D(*input: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence] = DEFAULT_POINT3D_VALUES*, *unit: beartype.typing.Optional[pint.Unit] = None*)

Bases: `numpy.ndarray`, *`ansys.geometry.core.misc.units.PhysicalQuantity`*

Provides geometry primitive representation for a 3D point.

Overview

Methods

unit	Get the unit of the object.
base_unit	Get the base unit of the object.
transform	Transform the 3D point with a transformation matrix.

Properties

x	X plane component value.
y	Y plane component value.
z	Z plane component value.

Special methods

__eq__	Equals operator for the `Point3D` class.
__ne__	Not equals operator for the `Point3D` class.
__add__	Add operation for the `Point3D` class.
__sub__	Subtraction operation for the `Point3D` class.

Import detail

```
from ansys.geometry.core.math.point import Point3D
```

Property detail

property Point3D.**x**: pint.Quantity

X plane component value.

property Point3D.**y**: pint.Quantity

Y plane component value.

property Point3D.**z**: pint.Quantity

Z plane component value.

Method detail

`Point3D.__eq__`(*other:* Point3D) → bool

Equals operator for the `Point3D` class.

`Point3D.__ne__`(*other:* Point3D) → bool

Not equals operator for the `Point3D` class.

`Point3D.__add__`(*other: beartype.typing.Union[*Point3D, ansys.geometry.core.math.vector.Vector3D*]*) → *Point3D*

Add operation for the `Point3D` class.

`Point3D.__sub__`(*other: beartype.typing.Union[*Point3D, ansys.geometry.core.math.vector.Vector3D*]*) → *Point3D*

Subtraction operation for the `Point3D` class.

`Point3D.unit`() → pint.Unit

Get the unit of the object.

`Point3D.base_unit`() → pint.Unit

Get the base unit of the object.

`Point3D.transform`(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Point3D*

Transform the 3D point with a transformation matrix.

Parameters

matrix
[*Matrix44*] 4x4 transformation matrix to apply to the point.

Returns

Point3D
New 3D point that is the transformed copy of the original 3D point after applying the transformation matrix.

Notes

Transform the `Point3D` object by applying the specified 4x4 transformation matrix and return a new `Point3D` object representing the transformed point.

Description

Provides geometry primitive representation for 2D and 3D points.

Module detail

`point.DEFAULT_POINT2D_VALUES`

Default values for a 2D point.

`point.DEFAULT_POINT3D_VALUES`

Default values for a 3D point.

`point.BASE_UNIT_LENGTH`

Default value for the length of the base unit.

The `vector.py` module

Summary

Classes

Vector3D	Provides for managing and creating a 3D vector.
Vector2D	Provides for creating and managing a 2D vector.
UnitVector3D	Provides for creating and managing a 3D unit vector.
UnitVector2D	Provides for creating and managing a 3D unit vector.

Vector3D

class Vector3D(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: `numpy.ndarray`

Provides for managing and creating a 3D vector.

Overview

Constructors

from_points	Create a 3D vector from two distinct 3D points.

Methods

is_perpendicular_to	Check if this vector and another vector are perpendicular.
is_parallel_to	Check if this vector and another vector are parallel.
is_opposite	Check if this vector and another vector are opposite.
normalize	Return a normalized version of the 3D vector.
transform	Transform the 3D vector3D with a transformation matrix.
get_angle_between	Get the angle between this 3D vector and another 3D vector.
cross	Get the cross product of `Vector3D` objects.

Properties

x	X coordinate of the `Vector3D` class.
y	Y coordinate of the `Vector3D` class.
z	Z coordinate of the `Vector3D` class.
norm	Norm of the vector.
magnitude	Norm of the vector.
is_zero	Check if all components of the 3D vector are zero.

Special methods

`__eq__`	Equals operator for the `Vector3D` class.
`__ne__`	Not equals operator for the `Vector3D` class.
`__mul__`	Overload * operator with dot product.
`__mod__`	Overload % operator with cross product.
`__add__`	Addition operation overload for 3D vectors.
`__sub__`	Subtraction operation overload for 3D vectors.

Import detail

```
from ansys.geometry.core.math.vector import Vector3D
```

Property detail

property `Vector3D.`**`x`**: **Real**

X coordinate of the `Vector3D` class.

property `Vector3D.`**`y`**: **Real**

Y coordinate of the `Vector3D` class.

property `Vector3D.`**`z`**: **Real**

Z coordinate of the `Vector3D` class.

property `Vector3D.`**`norm`**: `float`

Norm of the vector.

property `Vector3D.`**`magnitude`**: `float`

Norm of the vector.

property `Vector3D.`**`is_zero`**: `bool`

Check if all components of the 3D vector are zero.

Method detail

`Vector3D.`**`is_perpendicular_to`**(*other_vector:* Vector3D) → bool

Check if this vector and another vector are perpendicular.

`Vector3D.`**`is_parallel_to`**(*other_vector:* Vector3D) → bool

Check if this vector and another vector are parallel.

`Vector3D.`**`is_opposite`**(*other_vector:* Vector3D) → bool

Check if this vector and another vector are opposite.

`Vector3D.`**`normalize`**() → *Vector3D*

Return a normalized version of the 3D vector.

`Vector3D.`**`transform`**(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Vector3D*

Transform the 3D vector3D with a transformation matrix.

Parameters

matrix
[*Matrix44*] 4x4 transformation matrix to apply to the vector.

Returns

Vector3D
A new 3D vector that is the transformed copy of the original 3D vector after applying the transformation matrix.

Notes

Transform the `Vector3D` object by applying the specified 4x4 transformation matrix and return a new `Vector3D` object representing the transformed vector.

`Vector3D.`**`get_angle_between`**`(`*v:* Vector3D`)` → pint.Quantity

Get the angle between this 3D vector and another 3D vector.

Parameters

v
[*Vector3D*] Other 3D vector for computing the angle.

Returns

Quantity
Angle between these two 3D vectors.

`Vector3D.`**`cross`**`(`*v:* Vector3D`)` → *Vector3D*

Get the cross product of `Vector3D` objects.

`Vector3D.`**`__eq__`**`(`*other:* Vector3D`)` → bool

Equals operator for the `Vector3D` class.

`Vector3D.`**`__ne__`**`(`*other:* Vector3D`)` → bool

Not equals operator for the `Vector3D` class.

`Vector3D.`**`__mul__`**`(`*other: beartype.typing.Union[*Vector3D*, ansys.geometry.core.typing.Real]*`)` → beartype.typing.Union[*Vector3D*, ansys.geometry.core.typing.Real]

Overload * operator with dot product.

Notes

This method also admits scalar multiplication.

`Vector3D.`**`__mod__`**`(`*other:* Vector3D`)` → *Vector3D*

Overload % operator with cross product.

`Vector3D.`**`__add__`**`(`*other: beartype.typing.Union[*Vector3D, ansys.geometry.core.math.point.Point3D*]*`)` → beartype.typing.Union[*Vector3D*, *ansys.geometry.core.math.point.Point3D*]

Addition operation overload for 3D vectors.

`Vector3D.`**`__sub__`**`(`*other:* Vector3D`)` → *Vector3D*

Subtraction operation overload for 3D vectors.

classmethod Vector3D.**from_points**(*point_a: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*, *point_b: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*)

Create a 3D vector from two distinct 3D points.

Parameters

point_a
[*Point3D*] *Point3D* class representing the first point.

point_b
[*Point3D*] *Point3D* class representing the second point.

Returns

Vector3D
3D vector from `point_a` to `point_b`.

Notes

The resulting 3D vector is always expressed in `Point3D` base units.

Vector2D

class Vector2D(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: `numpy.ndarray`

Provides for creating and managing a 2D vector.

Overview

Constructors

from_points	Create a 2D vector from two distinct 2D points.

Methods

cross	Return the cross product of `Vector2D` objects.
is_perpendicular_to	Check if this 2D vector and another 2D vector are perpendicular.
is_parallel_to	Check if this vector and another vector are parallel.
is_opposite	Check if this vector and another vector are opposite.
normalize	Return a normalized version of the 2D vector.
get_angle_between	Get the angle between this 2D vector and another 2D vector.

Properties

x	X coordinate of the 2D vector.
y	Y coordinate of the 2D vector.
norm	Norm of the 2D vector.
magnitude	Norm of the 2D vector.
is_zero	Check if values for all components of the 2D vector are zero.

Special methods

__eq__	Equals operator for the `Vector2D` class.
__ne__	Not equals operator for the `Vector2D` class.
__mul__	Overload * operator with dot product.
__add__	Addition operation overload for 2D vectors.
__sub__	Subtraction operation overload for 2D vectors.
__mod__	Overload % operator with cross product.

Import detail

```
from ansys.geometry.core.math.vector import Vector2D
```

Property detail

property `Vector2D.`**`x: Real`**

X coordinate of the 2D vector.

property `Vector2D.`**`y: Real`**

Y coordinate of the 2D vector.

property `Vector2D.`**`norm:`** `float`

Norm of the 2D vector.

property `Vector2D.`**`magnitude:`** `float`

Norm of the 2D vector.

property `Vector2D.`**`is_zero:`** `bool`

Check if values for all components of the 2D vector are zero.

Method detail

`Vector2D.`**`cross`**(*v:* Vector2D)

Return the cross product of `Vector2D` objects.

`Vector2D.`**`is_perpendicular_to`**(*other_vector:* Vector2D) → bool

Check if this 2D vector and another 2D vector are perpendicular.

`Vector2D.`**`is_parallel_to`**`(`*other_vector:* Vector2D`)` → bool

Check if this vector and another vector are parallel.

`Vector2D.`**`is_opposite`**`(`*other_vector:* Vector2D`)` → bool

Check if this vector and another vector are opposite.

`Vector2D.`**`normalize`**`()` → *Vector2D*

Return a normalized version of the 2D vector.

`Vector2D.`**`get_angle_between`**`(`*v:* Vector2D`)` → pint.Quantity

Get the angle between this 2D vector and another 2D vector.

Parameters

v
[*`Vector2D`*] Other 2D vector to compute the angle with.

Returns

`Quantity`
Angle between these two 2D vectors.

`Vector2D.`**`__eq__`**`(`*other:* Vector2D`)` → bool

Equals operator for the `Vector2D` class.

`Vector2D.`**`__ne__`**`(`*other:* Vector2D`)` → bool

Not equals operator for the `Vector2D` class.

`Vector2D.`**`__mul__`**`(`*other: beartype.typing.Union[*Vector2D*, ansys.geometry.core.typing.Real]*`)` → beartype.typing.Union[*Vector2D*, ansys.geometry.core.typing.Real]

Overload * operator with dot product.

Notes

This method also admits scalar multiplication.

`Vector2D.`**`__add__`**`(`*other: beartype.typing.Union[*Vector2D, ansys.geometry.core.math.point.Point2D*]*`)` → beartype.typing.Union[*Vector2D*, *ansys.geometry.core.math.point.Point2D*]

Addition operation overload for 2D vectors.

`Vector2D.`**`__sub__`**`(`*other:* Vector2D`)` → *Vector2D*

Subtraction operation overload for 2D vectors.

`Vector2D.`**`__mod__`**`(`*other:* Vector2D`)` → *Vector2D*

Overload % operator with cross product.

`classmethod` `Vector2D.`**`from_points`**`(`*point_a: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point2D*], point_b: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point2D*]*`)`

Create a 2D vector from two distinct 2D points.

Parameters

point_a
[*`Point2D`*] *`Point2D`* class representing the first point.

point_b
[*Point2D*] *Point2D* class representing the second point.

Returns

Vector2D
2D vector from `point_a` to `point_b`.

Notes

The resulting 2D vector is always expressed in `Point2D` base units.

UnitVector3D

class UnitVector3D(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: *Vector3D*

Provides for creating and managing a 3D unit vector.

Overview

Constructors

from_points	Create a 3D unit vector from two distinct 3D points.

Import detail

```
from ansys.geometry.core.math.vector import UnitVector3D
```

Method detail

classmethod UnitVector3D.**from_points**(*point_a: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*, *point_b: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*)

Create a 3D unit vector from two distinct 3D points.

Parameters

point_a
[*Point3D*] *Point3D* class representing the first point.

point_b
[*Point3D*] *Point3D* class representing the second point.

Returns

UnitVector3D
3D unit vector from `point_a` to `point_b`.

UnitVector2D

class UnitVector2D(*shape*, *dtype=float*, *buffer=None*, *offset=0*, *strides=None*, *order=None*)

Bases: *Vector2D*

Provides for creating and managing a 3D unit vector.

Overview

Constructors

from_points	Create a 2D unit vector from two distinct 2D points.

Import detail

```
from ansys.geometry.core.math.vector import UnitVector2D
```

Method detail

classmethod UnitVector2D.**from_points**(*point_a: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point2D*]*, *point_b: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point2D*]*)

Create a 2D unit vector from two distinct 2D points.

Parameters

point_a
[*Point2D*] *Point2D* class representing the first point.

point_b
[*Point2D*] *Point2D* class representing the second point.

Returns

UnitVector2D
2D unit vector from `point_a` to `point_b`.

Description

Provides for creating and managing 2D and 3D vectors.

Description

PyAnsys Geometry math subpackage.

The `misc` package

Summary

Submodules

accuracy	Provides for evaluating decimal precision.
checks	Provides functions for performing common checks.
measurements	Provides various measurement-related classes.
options	Provides various option classes.
units	Provides for handling units homogeneously throughout PyAnsys Geometry.

The `accuracy.py` module

Summary

Classes

Accuracy	Provides decimal precision evaluations for actions such as equivalency.

Constants

LENGTH_ACCURACY	Constant for decimal accuracy in length comparisons.
ANGLE_ACCURACY	Constant for decimal accuracy in angle comparisons.

Accuracy

`class Accuracy`

Provides decimal precision evaluations for actions such as equivalency.

Overview

Methods

length_is_equal	Check if the comparison length is equal to the reference length.
length_is_greater_than_or_equal	Check if the comparison length is greater than the reference length.
length_is_less_than_or_equal	Check if the comparison length is less than or equal to the reference length.
length_is_zero	Check if the length is within the length accuracy of exact zero.
length_is_negative	Check if the length is below a negative length accuracy.
length_is_positive	Check if the length is above a positive length accuracy.
angle_is_zero	Check if the length is within the angle accuracy of exact zero.
angle_is_negative	Check if the angle is below a negative angle accuracy.
angle_is_positive	Check if the angle is above a positive angle accuracy.
is_within_tolerance	Check if two values (a and b) are inside a relative and absolute tolerance.

Import detail

```python
from ansys.geometry.core.misc.accuracy import Accuracy
```

Method detail

Accuracy.**length_is_equal**(*reference_length: ansys.geometry.core.typing.Real*) → bool

Check if the comparison length is equal to the reference length.

Returns

bool
`True` if the comparison length is equal to the reference length within the length accuracy, `False` otherwise.

Notes

The check is done up to the constant value specified for `LENGTH_ACCURACY`.

Accuracy.**length_is_greater_than_or_equal**(*reference_length: ansys.geometry.core.typing.Real*) → bool

Check if the comparison length is greater than the reference length.

Returns

bool
`True` if the comparison length is greater than the reference length within the length accuracy, `False` otherwise.

Notes

The check is done up to the constant value specified for `LENGTH_ACCURACY`.

`Accuracy.`**`length_is_less_than_or_equal`**`(`*reference_length: ansys.geometry.core.typing.Real*`)` → bool

Check if the comparison length is less than or equal to the reference length.

Returns

bool
`True` if the comparison length is less than or equal to the reference length within the length accuracy, `False` otherwise.

Notes

The check is done up to the constant value specified for `LENGTH_ACCURACY`.

`Accuracy.`**`length_is_zero`**`()` → bool

Check if the length is within the length accuracy of exact zero.

Returns

bool
`True` if the length is within the length accuracy of exact zero, `False` otherwise.

`Accuracy.`**`length_is_negative`**`()` → bool

Check if the length is below a negative length accuracy.

Returns

bool

`True` if the length is below a negative length accuracy,
`False` otherwise.

`Accuracy.`**`length_is_positive`**`()` → bool

Check if the length is above a positive length accuracy.

Returns

bool

`True` if the length is above a positive length accuracy,
`False` otherwise.

`Accuracy.`**`angle_is_zero`**`()` → bool

Check if the length is within the angle accuracy of exact zero.

Returns

bool

`True` if the length is within the angle accuracy of exact zero,
`False` otherwise.

`Accuracy.`**`angle_is_negative`**`()` → bool

Check if the angle is below a negative angle accuracy.

Returns

bool

`True` if the angle is below a negative angle accuracy,
`False` otherwise.

`Accuracy.`**`angle_is_positive`**`()` → bool

Check if the angle is above a positive angle accuracy.

Returns

bool

`True` if the angle is above a positive angle accuracy,
`False` otherwise.

`Accuracy.`**`is_within_tolerance`**`(`*b: ansys.geometry.core.typing.Real*, *relative_tolerance: ansys.geometry.core.typing.Real*, *absolute_tolerance: ansys.geometry.core.typing.Real*`)` → bool

Check if two values (a and b) are inside a relative and absolute tolerance.

Parameters

a
[`Real`] First value.

b
[`Real`] Second value.

relative_tolerance
[`Real`] Relative tolerance accepted.

absolute_tolerance
[`Real`] Absolute tolerance accepted.

Returns

bool
`True` if the values are inside the accepted tolerances, `False` otherwise.

Description

Provides for evaluating decimal precision.

Module detail

`accuracy.`**`LENGTH_ACCURACY = 1e-08`**

Constant for decimal accuracy in length comparisons.

`accuracy.`**`ANGLE_ACCURACY = 1e-06`**

Constant for decimal accuracy in angle comparisons.

The `checks.py` module

Summary

Functions

check_is_float_int	Check if a parameter has a float or integer value.
check_ndarray_is_float_int	Check if a `numpy.ndarray` has float or integer values.
check_ndarray_is_not_none	Check if a `numpy.ndarray` has all `None` values.
check_ndarray_is_all_nan	Check if a `numpy.ndarray` is all nan-valued.
check_ndarray_is_non_zero	Check if a `numpy.ndarray` is zero-valued.
check_pint_unit_compatibility	Check if input for `pint.Unit` is compatible with the expected input.
check_type_equivalence	Check if an input object is of the same class as an expected object.
check_type	Check if an input object is of the same type as expected types.

Description

Provides functions for performing common checks.

Module detail

`checks.`**`check_is_float_int`**(*param: object, param_name: beartype.typing.Optional[beartype.typing.Union[str, None]] = None*) → None

Check if a parameter has a float or integer value.

Parameters

param
[`object`] Object instance to check.

param_name
[`str`, default: `None`] Parameter name (if any).

Raises

`TypeError`
If the parameter does not have a float or integer value.

`checks.`**`check_ndarray_is_float_int`**(*param: numpy.ndarray, param_name: beartype.typing.Optional[beartype.typing.Union[str, None]] = None*) → None

Check if a `numpy.ndarray` has float or integer values.

Parameters

param
[`ndarray`] `numpy.ndarray` instance to check.

param_name
[`str`, default: `None`] `numpy.ndarray` instance name (if any).

Raises

`TypeError`
If the `numpy.ndarray` instance does not have float or integer values.

`checks.`**`check_ndarray_is_not_none`**(*param: numpy.ndarray*, *param_name: beartype.typing.Optional[beartype.typing.Union[str, None]] = None*) → None

Check if a `numpy.ndarray` has all `None` values.

Parameters

param
[`ndarray`] `numpy.ndarray` instance to check.

param_name
[`str`, default: `None`] `numpy.ndarray` instance name (if any).

Raises

`ValueError`
If the `numpy.ndarray` instance has a value of `None` for all parameters.

`checks.`**`check_ndarray_is_all_nan`**(*param: numpy.ndarray*, *param_name: beartype.typing.Optional[beartype.typing.Union[str, None]] = None*) → None

Check if a `numpy.ndarray` is all nan-valued.

Parameters

param
[`ndarray`] `numpy.ndarray` instance to check.

param_name
[`str` or `None`, default: `None`] `numpy.ndarray` instance name (if any).

Raises

`ValueError`
If the `numpy.ndarray` instance is all nan-valued.

`checks.`**`check_ndarray_is_non_zero`**(*param: numpy.ndarray*, *param_name: beartype.typing.Optional[beartype.typing.Union[str, None]] = None*) → None

Check if a `numpy.ndarray` is zero-valued.

Parameters

param
[`ndarray`] `numpy.ndarray` instance to check.

param_name
[`str`, default: `None`] `numpy.ndarray` instance name (if any).

Raises

`ValueError`
If the `numpy.ndarray` instance is zero-valued.

`checks.`**`check_pint_unit_compatibility`**(*input: pint.Unit*, *expected: pint.Unit*) → None

Check if input for `pint.Unit` is compatible with the expected input.

Parameters

input
[`Unit`] `pint.Unit` input.

expected
[`Unit`] `pint.Unit` expected dimensionality.

Raises

`TypeError`
If the input is not compatible with the `pint.Unit` class.

`checks.`**`check_type_equivalence`**(*input: object*, *expected: object*) → None

Check if an input object is of the same class as an expected object.

Parameters

input
[`object`] Input object.

expected
[`object`] Expected object.

Raises

`TypeError`
If the objects are not of the same class.

`checks.`**`check_type`**(*input: object*, *expected_type: beartype.typing.Union[type, beartype.typing.Tuple[type, beartype.typing.Any]]*) → None

Check if an input object is of the same type as expected types.

Parameters

input
[`object`] Input object.

expected_type
[`Union`[`type`, `Tuple`[`type`, ...]]] One or more types to compare the input object against.

Raises

`TypeError`
If the object does not match the one or more expected types.

The `measurements.py` module

Summary

Classes

`SingletonMeta`	Provides a thread-safe implementation of a singleton design pattern.
`DefaultUnitsClass`	Provides default units for the PyAnsys Geometry singleton design pattern.
`Measurement`	Provides the `PhysicalQuantity` subclass for holding a measurement.
`Distance`	Provides the `Measurement` subclass for holding a distance.
`Angle`	Provides the `Measurement` subclass for holding an angle.

Constants

DEFAULT_UNITS	PyAnsys Geometry default units object.

SingletonMeta

class SingletonMeta

Bases: `type`

Provides a thread-safe implementation of a singleton design pattern.

Overview

Special methods

__call__	Return a single instance of the class.

Import detail

```
from ansys.geometry.core.misc.measurements import SingletonMeta
```

Method detail

`SingletonMeta.`**`__call__`**`(*args, **kwargs)`

Return a single instance of the class.

Possible changes to the value of the `__init__` argument do not affect the returned instance.

DefaultUnitsClass

class DefaultUnitsClass

Provides default units for the PyAnsys Geometry singleton design pattern.

Overview

Properties

LENGTH	Default length unit for PyAnsys Geometry.
ANGLE	Default angle unit for PyAnsys Geometry.
SERVER_LENGTH	Default length unit for supporting Geometry services for gRPC messages.
SERVER_AREA	Default area unit for supporting Geometry services for gRPC messages.
SERVER_VOLUME	Default volume unit for supporting Geometry services for gRPC messages.
SERVER_ANGLE	Default angle unit for supporting Geometry services for gRPC messages.

Import detail

```
from ansys.geometry.core.misc.measurements import DefaultUnitsClass
```

Property detail

property DefaultUnitsClass.**LENGTH**: pint.Unit

Default length unit for PyAnsys Geometry.

property DefaultUnitsClass.**ANGLE**: pint.Unit

Default angle unit for PyAnsys Geometry.

property DefaultUnitsClass.**SERVER_LENGTH**: pint.Unit

Default length unit for supporting Geometry services for gRPC messages.

Notes

The default units on the server side are not modifiable yet.

property DefaultUnitsClass.**SERVER_AREA**: pint.Unit

Default area unit for supporting Geometry services for gRPC messages.

Notes

The default units on the server side are not modifiable yet.

property DefaultUnitsClass.**SERVER_VOLUME**: pint.Unit

Default volume unit for supporting Geometry services for gRPC messages.

Notes

The default units on the server side are not modifiable yet.

property DefaultUnitsClass.**SERVER_ANGLE**: pint.Unit

Default angle unit for supporting Geometry services for gRPC messages.

Notes

The default units on the server side are not modifiable yet.

Measurement

class Measurement(*value: beartype.typing.Union[ansys.geometry.core.typing.Real, pint.Quantity], unit: pint.Unit, dimensions: pint.Unit*)

Bases: *ansys.geometry.core.misc.units.PhysicalQuantity*

Provides the PhysicalQuantity subclass for holding a measurement.

Overview

Properties

value	Value of the measurement.

Special methods

__eq__	Equals operator for the `Measurement` class.

Import detail

```
from ansys.geometry.core.misc.measurements import Measurement
```

Property detail

property Measurement.**value**: pint.Quantity

Value of the measurement.

Method detail

Measurement.**__eq__**(*other:* Measurement) → bool

Equals operator for the `Measurement` class.

Distance

class Distance(*value: beartype.typing.Union[ansys.geometry.core.typing.Real, pint.Quantity], unit: beartype.typing.Optional[pint.Unit] = None*)

Bases: *Measurement*

Provides the `Measurement` subclass for holding a distance.

Import detail

```
from ansys.geometry.core.misc.measurements import Distance
```

Angle

class Angle(*value: beartype.typing.Union[ansys.geometry.core.typing.Real, pint.Quantity], unit: beartype.typing.Optional[pint.Unit] = None*)

Bases: *Measurement*

Provides the `Measurement` subclass for holding an angle.

Import detail

```
from ansys.geometry.core.misc.measurements import Angle
```

Description

Provides various measurement-related classes.

Module detail

measurements.**DEFAULT_UNITS**

PyAnsys Geometry default units object.

The `options.py` module

Summary

Classes

ImportOptions	Import options when opening a file.

ImportOptions

class ImportOptions

Import options when opening a file.

Overview

Methods

to_dict	Provide the dictionary representation of the ImportOptions class.

Attributes

cleanup_bodies
import_coordinate_systems
import_curves
import_hidden_components_and_geometry
import_names
import_planes
import_points

Import detail

```
from ansys.geometry.core.misc.options import ImportOptions
```

Attribute detail

ImportOptions.**cleanup_bodies**: bool = False

ImportOptions.**import_coordinate_systems**: bool = False

ImportOptions.**import_curves**: bool = False

ImportOptions.**import_hidden_components_and_geometry**: bool = False

ImportOptions.**import_names**: bool = False

ImportOptions.**import_planes**: bool = False

ImportOptions.**import_points**: bool = False

Method detail

ImportOptions.**to_dict**()

Provide the dictionary representation of the ImportOptions class.

Description

Provides various option classes.

The `units.py` module

Summary

Classes

PhysicalQuantity	Provides the base class for handling units throughout PyAnsys Geometry.

Constants

UNITS	Units manager.

PhysicalQuantity

class PhysicalQuantity(*unit: pint.Unit, expected_dimensions: beartype.typing.Optional[pint.Unit] = None*)

Provides the base class for handling units throughout PyAnsys Geometry.

Overview

Properties

unit	Unit of the object.
base_unit	Base unit of the object.

Import detail

```python
from ansys.geometry.core.misc.units import PhysicalQuantity
```

Property detail

property PhysicalQuantity.**unit: pint.Unit**

Unit of the object.

property PhysicalQuantity.**base_unit: pint.Unit**

Base unit of the object.

Description

Provides for handling units homogeneously throughout PyAnsys Geometry.

Module detail

`units.UNITS`

Units manager.

Description

Provides the PyAnsys Geometry miscellaneous subpackage.

The `plotting` package

Summary

Subpackages

widgets	Submodule providing widgets for the PyAnsys Geometry plotter.

Submodules

plotter	Provides plotting for various PyAnsys Geometry objects.
plotter_helper	Provides a wrapper to aid in plotting.
plotting_types	Data types for plotting.
trame_gui	Module for using trame for visualization.

The `widgets` package

Summary

Submodules

button	Provides for implementing buttons in PyAnsys Geometry.
displace_arrows	Provides the displacement arrows widget for the PyVista plotter.
measure	Provides the ruler widget for the PyAnsys Geometry plotter.
ruler	Provides the ruler widget for the PyAnsys Geometry plotter.
show_design_point	Provides the ruler widget for the PyAnsys Geometry plotter.
view_button	Provides the view button widget for changing the camera view.
widget	Provides the abstract implementation of plotter widgets.

The `button.py` module

Summary

Classes

Button	Provides the abstract class for implementing buttons in PyAnsys Geometry.

Button

class Button(*plotter: pyvista.Plotter*, *button_config: tuple*)

Bases: *ansys.geometry.core.plotting.widgets.widget.PlotterWidget*

Provides the abstract class for implementing buttons in PyAnsys Geometry.

Overview

Abstract methods

callback	Get the functionality of the button, which is implemented by subclasses.

Methods

update	Assign the image that represents the button.

Import detail

```
from ansys.geometry.core.plotting.widgets.button import Button
```

Method detail

abstract Button.**callback**(*state: bool*) → None

Get the functionality of the button, which is implemented by subclasses.

Parameters

state
[bool] Whether the button is active.

Button.**update**() → None

Assign the image that represents the button.

Description

Provides for implementing buttons in PyAnsys Geometry.

The `displace_arrows.py` module

Summary

Classes

DisplacementArrow	Defines the arrow to draw and what it is to do.

Enums

CameraPanDirection	Provides an enum with the available movement directions of the camera.

DisplacementArrow

class DisplacementArrow(*plotter: pyvista.Plotter*, *direction:* CameraPanDirection)

Bases: *ansys.geometry.core.plotting.widgets.button.Button*

Defines the arrow to draw and what it is to do.

Overview

Methods

callback	Move the camera in the direction defined by the button.

Import detail

```
from ansys.geometry.core.plotting.widgets.displace_arrows import DisplacementArrow
```

Method detail

`DisplacementArrow.callback(state: bool) → None`

Move the camera in the direction defined by the button.

Parameters

state
[bool] State of the button, which is inherited from PyVista. The value is `True` if the button is active. However, this parameter is unused by this `callback` method.

CameraPanDirection

`class CameraPanDirection`

Bases: `enum.Enum`

Provides an enum with the available movement directions of the camera.

Overview

Attributes

XUP
XDOWN
YUP
YDOWN
ZUP
ZDOWN

Import detail

```
from ansys.geometry.core.plotting.widgets.displace_arrows import CameraPanDirection
```

Attribute detail

`CameraPanDirection.XUP = (0, 'upxarrow.png', (5, 170))`

`CameraPanDirection.XDOWN = (1, 'downarrow.png', (5, 130))`

`CameraPanDirection.YUP = (2, 'upyarrow.png', (35, 170))`

`CameraPanDirection.YDOWN = (3, 'downarrow.png', (35, 130))`

`CameraPanDirection.ZUP = (4, 'upzarrow.png', (65, 170))`

`CameraPanDirection.ZDOWN = (5, 'downarrow.png', (65, 130))`

Description

Provides the displacement arrows widget for the PyVista plotter.

The `measure.py` module

Summary

Classes

MeasureWidget	Provides the measure widget for the PyAnsys Geometry `Plotter` class.

MeasureWidget

class MeasureWidget(*plotter_helper:* PlotterHelper)

Bases: *ansys.geometry.core.plotting.widgets.widget.PlotterWidget*

Provides the measure widget for the PyAnsys Geometry `Plotter` class.

Overview

Methods

callback	Remove or add the measurement widget actor upon click.
update	Define the measurement widget button params.

Import detail

```
from ansys.geometry.core.plotting.widgets.measure import MeasureWidget
```

Method detail

MeasureWidget.**callback**(*state: bool*) → None

Remove or add the measurement widget actor upon click.

Parameters

state

[bool] State of the button, which is inherited from PyVista. The value is `True` if the button is active.

MeasureWidget.**update**() → None

Define the measurement widget button params.

Description

Provides the ruler widget for the PyAnsys Geometry plotter.

The `ruler.py` module

Summary

Classes

`Ruler`	Provides the ruler widget for the PyAnsys Geometry `Plotter` class.

Ruler

class Ruler(*plotter: pyvista.Plotter*)

Bases: *`ansys.geometry.core.plotting.widgets.widget.PlotterWidget`*

Provides the ruler widget for the PyAnsys Geometry `Plotter` class.

Overview

Methods

`callback`	Remove or add the ruler widget actor upon click.
`update`	Define the configuration and representation of the ruler widget button.

Import detail

```
from ansys.geometry.core.plotting.widgets.ruler import Ruler
```

Method detail

`Ruler.`**`callback`**(*state: bool*) → None

Remove or add the ruler widget actor upon click.

Parameters

state

[bool] State of the button, which is inherited from PyVista. The value is `True` if the button is active.

Notes

This method provides a callback function for the ruler widet. It is called every time the ruler widget is clicked.

`Ruler.update()` → None

Define the configuration and representation of the ruler widget button.

Description

Provides the ruler widget for the PyAnsys Geometry plotter.

The `show_design_point.py` module

Summary

Classes

ShowDesignPoints	Provides the a button to hide/show DesignPoint objects in the plotter.

ShowDesignPoints

class ShowDesignPoints(*plotter_helper:* PlotterHelper)

Bases: *ansys.geometry.core.plotting.widgets.widget.PlotterWidget*

Provides the a button to hide/show DesignPoint objects in the plotter.

Overview

Methods

callback	Remove or add the DesignPoint actors upon click.
update	Define the configuration and representation of the button widget button.

Import detail

```
from ansys.geometry.core.plotting.widgets.show_design_point import ShowDesignPoints
```

Method detail

`ShowDesignPoints.`**`callback`**(*state: bool*) → None

Remove or add the DesignPoint actors upon click.

Parameters

state
[bool] State of the button, which is inherited from PyVista. The value is `True` if the button is active.

`ShowDesignPoints.`**`update`**`()` → None

Define the configuration and representation of the button widget button.

Description

Provides the ruler widget for the PyAnsys Geometry plotter.

The `view_button.py` module

Summary

Classes

`ViewButton`	Provides for changing the view.

Enums

`ViewDirection`	Provides an enum with the available views.

ViewButton

`class ViewButton`(*plotter: pyvista.Plotter*, *direction: tuple*)

Bases: *`ansys.geometry.core.plotting.widgets.button.Button`*

Provides for changing the view.

Overview

Methods

`callback`	Change the view depending on button interaction.

Import detail

```
from ansys.geometry.core.plotting.widgets.view_button import ViewButton
```

Method detail

`ViewButton.`**`callback`**(*state: bool*) → None

Change the view depending on button interaction.

Parameters

state
[bool] State of the button, which is inherited from PyVista. The value is `True` if the button is active.

Raises

`NotImplementedError`
Raised if the specified direction is not implemented.

ViewDirection

`class ViewDirection`

Bases: `enum.Enum`

Provides an enum with the available views.

Overview

Attributes

XYPLUS
XYMINUS
XZPLUS
XZMINUS
YZPLUS
YZMINUS
ISOMETRIC

Import detail

```
from ansys.geometry.core.plotting.widgets.view_button import ViewDirection
```

Attribute detail

`ViewDirection.XYPLUS = (0, '+xy.png', (5, 220))`

`ViewDirection.XYMINUS = (1, '-xy.png', (5, 251))`

`ViewDirection.XZPLUS = (2, '+xz.png', (5, 282))`

`ViewDirection.XZMINUS = (3, '-xz.png', (5, 313))`

`ViewDirection.YZPLUS = (4, '+yz.png', (5, 344))`

`ViewDirection.YZMINUS = (5, '-yz.png', (5, 375))`

`ViewDirection.ISOMETRIC = (6, 'isometric.png', (5, 406))`

Description

Provides the view button widget for changing the camera view.

The `widget.py` module

Summary

Classes

`PlotterWidget`	Provides an abstract class for plotter widgets.

PlotterWidget

`class PlotterWidget`(*plotter: pyvista.Plotter*)

Bases: `abc.ABC`

Provides an abstract class for plotter widgets.

Overview

Abstract methods

`callback`	General callback function for `PlotterWidget` objects.
`update`	General update function for `PlotterWidget` objects.

Properties

plotter	Plotter object the widget is assigned to.

Import detail

```
from ansys.geometry.core.plotting.widgets.widget import PlotterWidget
```

Property detail

property PlotterWidget.**plotter**: pyvista.Plotter

Plotter object the widget is assigned to.

Method detail

abstract PlotterWidget.**callback**(*state*) → None

General callback function for `PlotterWidget` objects.

abstract PlotterWidget.**update**() → None

General update function for `PlotterWidget` objects.

Description

Provides the abstract implementation of plotter widgets.

Description

Submodule providing widgets for the PyAnsys Geometry plotter.

The `plotter.py` module

Summary

Classes

Plotter	Provides for plotting sketches and bodies.

Constants

DEFAULT_COLOR	Default color we use for the plotter actors.
PICKED_COLOR	Color to use for the actors that are currently picked.
EDGE_COLOR	Default color to use for the edges.
PICKED_EDGE_COLOR	Color to use for the edges that are currently picked.

Plotter

class Plotter(*scene: beartype.typing.Optional[pyvista.Plotter] = None*, *color_opts: beartype.typing.Optional[beartype.typing.Dict] = None*, *num_points: int = 100*, *enable_widgets: bool = True*)

Provides for plotting sketches and bodies.

Overview

Methods

view_xy	View the scene from the XY plane.
view_xz	View the scene from the XZ plane.
view_yx	View the scene from the YX plane.
view_yz	View the scene from the YZ plane.
view_zx	View the scene from the ZX plane.
view_zy	View the scene from the ZY plane.
plot_frame	Plot a frame in the scene.
plot_plane	Plot a plane in the scene.
plot_sketch	Plot a sketch in the scene.
add_body_edges	Add the outer edges of a body to the plot.
add_body	Add a body to the scene.
add_component	Add a component to the scene.
add_sketch_polydata	Add sketches to the scene from PyVista polydata.
add_design_point	Add a DesignPoint object to the plotter.
add	Add any type of object to the scene.
add_list	Add a list of any type of object to the scene.
show	Show the rendered scene on the screen.

Properties

scene	Rendered scene object.

Import detail

```
from ansys.geometry.core.plotting.plotter import Plotter
```

Property detail

property Plotter.**scene**: pyvista.plotting.plotter.Plotter

Rendered scene object.

Returns

`Plotter`
Rendered scene object.

Method detail

Plotter.**view_xy**() → None

View the scene from the XY plane.

Plotter.**view_xz**() → None

View the scene from the XZ plane.

Plotter.**view_yx**() → None

View the scene from the YX plane.

Plotter.**view_yz**() → None

View the scene from the YZ plane.

Plotter.**view_zx**() → None

View the scene from the ZX plane.

Plotter.**view_zy**() → None

View the scene from the ZY plane.

Plotter.**plot_frame**(*frame:* ansys.geometry.core.math.frame.Frame, *plotting_options: beartype.typing.Optional[beartype.typing.Dict] = None*) → None

Plot a frame in the scene.

Parameters

frame
[*Frame*] Frame to render in the scene.

plotting_options
[`dict`, default: `None`] Dictionary containing parameters accepted by the `pyvista.create_axes_marker()` class for customizing the frame rendering in the scene.

Plotter.**plot_plane**(*plane:* ansys.geometry.core.math.plane.Plane, *plane_options: beartype.typing.Optional[beartype.typing.Dict] = None*, *plotting_options: beartype.typing.Optional[beartype.typing.Dict] = None*) → None

Plot a plane in the scene.

Parameters

plane
[*Plane*] Plane to render in the scene.

plane_options
[`dict`, default: `None`] Dictionary containing parameters accepted by the `pyvista.Plane` function for customizing the mesh representing the plane.

plotting_options
[`dict`, default: `None`] Dictionary containing parameters accepted by the `Plotter.add_mesh` method for customizing the mesh rendering of the plane.

`Plotter.`**`plot_sketch`**`(`*sketch:* ansys.geometry.core.sketch.sketch.Sketch, *show_plane: bool = False*, *show_frame: bool = False*, ***plotting_options: beartype.typing.Optional[beartype.typing.Dict]*`)` → None

Plot a sketch in the scene.

Parameters

sketch
[*`Sketch`*] Sketch to render in the scene.

show_plane
[bool, default: `False`] Whether to render the sketch plane in the scene.

show_frame
[bool, default: `False`] Whether to show the frame in the scene.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`Plotter.`**`add_body_edges`**`(`*body_plot:* ansys.geometry.core.plotting.plotting_types.GeomObjectPlot, ***plotting_options: beartype.typing.Optional[dict]*`)` → None

Add the outer edges of a body to the plot.

This method has the side effect of adding the edges to the GeomObject that you pass through the parameters.

Parameters

body
[*`GeomObjectPlot`*] Body of which to add the edges.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`Plotter.`**`add_body`**`(`*body:* ansys.geometry.core.designer.body.Body, *merge: beartype.typing.Optional[bool] = False*, ***plotting_options: beartype.typing.Optional[beartype.typing.Dict]*`)` → None

Add a body to the scene.

Parameters

body
[*`Body`*] Body to add.

merge
[bool, default: `False`] Whether to merge the body into a single mesh. When `True`, the individual faces of the tessellation are merged. This preserves the number of triangles and only merges the topology.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`Plotter.add_component`(*component:* ansys.geometry.core.designer.component.Component, *merge_component: bool = False*, *merge_bodies: bool = False*, ***plotting_options*) → str

Add a component to the scene.

Parameters

component
[`Component`] Component to add.

merge_component
[bool, default: `False`] Whether to merge the component into a single dataset. When `True`, all the individual bodies are effectively combined into a single dataset without any hierarchy.

merge_bodies
[bool, default: `False`] Whether to merge each body into a single dataset. When `True`, all the faces of each individual body are effectively combined into a single dataset without separating faces.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

Returns

`str`
Name of the added PyVista actor.

`Plotter.add_sketch_polydata`(*polydata_entries: beartype.typing.List[pyvista.PolyData]*, ***plotting_options*) → None

Add sketches to the scene from PyVista polydata.

Parameters

polydata
[`pyvista.PolyData`] Polydata to add.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`Plotter.add_design_point`(*design_point:* ansys.geometry.core.designer.designpoint.DesignPoint, ***plotting_options*) → None

Add a DesignPoint object to the plotter.

Parameters

design_point
[`DesignPoint`] DesignPoint to add.

`Plotter.add`(*object: beartype.typing.Any*, *merge_bodies: bool = False*, *merge_components: bool = False*, *filter: str = None*, ***plotting_options*) → beartype.typing.Dict[pyvista.Actor, *ansys.geometry.core.plotting.plotting_types.GeomObjectPlot*]

Add any type of object to the scene.

These types of objects are supported: `Body`, `Component`, `List[pv.PolyData]`, `pv.MultiBlock`, and `Sketch`.

Parameters

plotting_list
[`List[Any]`] List of objects that you want to plot.

merge_bodies
[bool, default: `False`] Whether to merge each body into a single dataset. When `True`, all the faces of each individual body are effectively combined into a single dataset without separating faces.

merge_component
[bool, default: `False`] Whether to merge the component into a single dataset. When `True`, all the individual bodies are effectively combined into a single dataset without any hierarchy.

filter
[`str`, default: `None`] Regular expression with the desired name or names you want to include in the plotter.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

Returns

`Dict[Actor, GeomObjectPlot]`
Mapping between the ~pyvista.Actor and the PyAnsys Geometry object.

`Plotter.add_list`(*plotting_list: beartype.typing.List[beartype.typing.Any]*, *merge_bodies: bool = False*, *merge_components: bool = False*, *filter: str = None*, ***plotting_options*) → beartype.typing.Dict[pyvista.Actor, *ansys.geometry.core.plotting.plotting_types.GeomObjectPlot*]

Add a list of any type of object to the scene.

These types of objects are supported: `Body`, `Component`, `List[pv.PolyData]`, `pv.MultiBlock`, and `Sketch`.

Parameters

plotting_list
[`List[Any]`] List of objects you want to plot.

merge_component
[bool, default: `False`] Whether to merge the component into a single dataset. When `True`, all the individual bodies are effectively combined into a single dataset without any hierarchy.

merge_bodies
[bool, default: `False`] Whether to merge each body into a single dataset. When `True`, all the faces of each individual body are effectively combined into a single dataset without separating faces.

filter
[`str`, default: `None`] Regular expression with the desired name or names you want to include in the plotter.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

Returns

`Dict[Actor, GeomObjectPlot]`
Mapping between the ~pyvista.Actor and the PyAnsys Geometry objects.

`Plotter.show`(*show_axes_at_origin: bool = True*, *show_plane: bool = True*, *jupyter_backend: beartype.typing.Optional[str] = None*, ***kwargs: beartype.typing.Optional[beartype.typing.Dict]*) → None

Show the rendered scene on the screen.

Parameters

jupyter_backend
[`str`, default: `None`] PyVista Jupyter backend.

****kwargs**
[`dict`, default: `None`] Plotting keyword arguments. For allowable keyword arguments, see the `Plotter.show` method.

Notes

For more information on supported Jupyter backends, see Jupyter Notebook Plotting in the PyVista documentation.

Description

Provides plotting for various PyAnsys Geometry objects.

Module detail

`plotter.DEFAULT_COLOR = '#D6F7D1'`
Default color we use for the plotter actors.

`plotter.PICKED_COLOR = '#BB6EEE'`
Color to use for the actors that are currently picked.

`plotter.EDGE_COLOR = '#000000'`
Default color to use for the edges.

`plotter.PICKED_EDGE_COLOR = '#9C9C9C'`
Color to use for the edges that are currently picked.

The `plotter_helper.py` module

Summary

Classes

PlotterHelper	Provides for simplifying the selection of trame in `plot()` functions.

PlotterHelper

class PlotterHelper(*use_trame: beartype.typing.Optional[bool] = None*, *allow_picking: beartype.typing.Optional[bool] = False*)

Provides for simplifying the selection of trame in `plot()` functions.

Overview

Methods

enable_widgets	Enable the widgets for the plotter.
select_object	Select an object in the plotter.
unselect_object	Unselect an object in the plotter.
picker_callback	Define callback for the element picker.
compute_edge_object_map	Compute the mapping between plotter actors and EdgePlot objects.
enable_picking	Enable picking capabilities in the plotter.
disable_picking	Disable picking capabilities in the plotter.
plot	Plot and show any PyAnsys Geometry object.
show_plotter	Show the plotter or start the trame service.

Import detail

```
from ansys.geometry.core.plotting.plotter_helper import PlotterHelper
```

Method detail

PlotterHelper.**enable_widgets**()

Enable the widgets for the plotter.

PlotterHelper.**select_object**(*geom_object: beartype.typing.Union[*ansys.geometry.core.plotting.plotting_types.GeomObjectPlot, ansys.geometry.core.plotting.plotting_types.EdgePlot*]*, *pt: numpy.ndarray*) → None

Select an object in the plotter.

Highlights the object edges and adds a label with the object name and adds it to the PyAnsys Geometry object selection.

Parameters

geom_object
[Union[*GeomObjectPlot*, *EdgePlot*]] Geometry object to select.

pt
[ndarray] Set of points to determine the label position.

PlotterHelper.**unselect_object**(*geom_object: beartype.typing.Union[*ansys.geometry.core.plotting.plotting_types.GeomObjectPlot, ansys.geometry.core.plotting.plotting_types.EdgePlot*]*) → None

Unselect an object in the plotter.

Removes edge highlighting and label from a plotter actor and removes it from the PyAnsys Geometry object selection.

Parameters

geom_object
[`Union`[*`GeomObjectPlot`*, *`EdgePlot`*]] Object to unselect.

`PlotterHelper.`**`picker_callback`**(*actor: pyvista.Actor*) → None

Define callback for the element picker.

Parameters

actor
[`Actor`] Actor that we are picking.

`PlotterHelper.`**`compute_edge_object_map`**() → beartype.typing.Dict[pyvista.Actor, *ansys.geometry.core.plotting.plotting_types.EdgePlot*]

Compute the mapping between plotter actors and EdgePlot objects.

Returns

`Dict[Actor, EdgePlot]`
Mapping between plotter actors and EdgePlot objects.

`PlotterHelper.`**`enable_picking`**()

Enable picking capabilities in the plotter.

`PlotterHelper.`**`disable_picking`**()

Disable picking capabilities in the plotter.

`PlotterHelper.`**`plot`**(*object: beartype.typing.Any*, *screenshot: beartype.typing.Optional[str] = None*, *merge_bodies: bool = False*, *merge_component: bool = False*, *view_2d: beartype.typing.Dict = None*, *filter: str = None*, ***plotting_options*) → beartype.typing.List[beartype.typing.Any]

Plot and show any PyAnsys Geometry object.

These types of objects are supported: `Body`, `Component`, `List[pv.PolyData]`, `pv.MultiBlock`, and `Sketch`.

Parameters

object
[`Any`] Any object or list of objects that you want to plot.

screenshot
[`str`, default: `None`] Path for saving a screenshot of the image that is being represented.

merge_bodies
[bool, default: `False`] Whether to merge each body into a single dataset. When `True`, all the faces of each individual body are effectively combined into a single dataset without separating faces.

merge_component
[bool, default: `False`] Whether to merge this component into a single dataset. When `True`, all the individual bodies are effectively combined into a single dataset without any hierarchy.

view_2d
[`Dict`, default: `None`] Dictionary with the plane and the viewup vectors of the 2D plane.

filter
[`str`, default: `None`] Regular expression with the desired name or names you want to include in the plotter.

****plotting_options**
[`dict`, default: `None`] Keyword arguments. For allowable keyword arguments, see the `Plotter.add_mesh` method.

Returns

`List[Any]`
List with the picked bodies in the picked order.

`PlotterHelper.`**`show_plotter`**(*screenshot: beartype.typing.Optional[str] = None*) → None

Show the plotter or start the trame service.

Parameters

plotter
[*`Plotter`*] PyAnsys Geometry plotter with the meshes added.

screenshot
[`str`, default: `None`] Path for saving a screenshot of the image that is being represented.

Description

Provides a wrapper to aid in plotting.

The `plotting_types.py` module

Summary

Classes

`EdgePlot`	Mapper class to relate PyAnsys Geometry edges with its PyVista actor.
`GeomObjectPlot`	Mapper class to relate PyAnsys Geometry objects with its PyVista actor.

EdgePlot

`class EdgePlot`(*actor: pyvista.Actor*, *edge_object:* ansys.geometry.core.designer.edge.Edge, *parent:* GeomObjectPlot = *None*)

Mapper class to relate PyAnsys Geometry edges with its PyVista actor.

Overview

Properties

actor	Return PyVista actor of the object.
edge_object	Return the PyAnsys Geometry edge.
parent	Parent PyAnsys Geometry object of this edge.
name	Return the name of the edge.

Import detail

```
from ansys.geometry.core.plotting.plotting_types import EdgePlot
```

Property detail

property EdgePlot.**actor**: pyvista.Actor

Return PyVista actor of the object.

Returns

Actor
: PyVista actor.

property EdgePlot.**edge_object**: ***Edge***

Return the PyAnsys Geometry edge.

Returns

Edge
: PyAnsys Geometry edge.

property EdgePlot.**parent**: **beartype.typing.Any**

Parent PyAnsys Geometry object of this edge.

Returns

Any
: PyAnsys Geometry object.

property EdgePlot.**name**: str

Return the name of the edge.

Returns

str
: Name of the edge.

GeomObjectPlot

class GeomObjectPlot(*actor: pyvista.Actor*, *object: beartype.typing.Any*, *edges: beartype.typing.List[*EdgePlot*] = None*, *add_body_edges: bool = True*)

Mapper class to relate PyAnsys Geometry objects with its PyVista actor.

Overview

Properties

actor	Return the PyVista actor of the PyAnsys Geometry object.
object	Return the PyAnsys Geometry object.
edges	Return the list of edges associated to this PyAnsys Geometry object.
name	Return the name of this object.
add_body_edges	Return whether you want to be able to add edges.

Import detail

```
from ansys.geometry.core.plotting.plotting_types import GeomObjectPlot
```

Property detail

property GeomObjectPlot.**actor: pyvista.Actor**

Return the PyVista actor of the PyAnsys Geometry object.

Returns

Actor
Actor of the PyAnsys Geometry object.

property GeomObjectPlot.**object: beartype.typing.Any**

Return the PyAnsys Geometry object.

Returns

Any
PyAnsys Geometry object.

property GeomObjectPlot.**edges: beartype.typing.List[*EdgePlot*]**

Return the list of edges associated to this PyAnsys Geometry object.

Returns

List[*EdgePlot*]
List of the edges of this object.

property GeomObjectPlot.**name: str**

Return the name of this object.

Returns

`str`
: Name of the object.

property `GeomObjectPlot.`**`add_body_edges:`** `bool`

Return whether you want to be able to add edges.

Returns

bool
: Flag to add edges.

Description

Data types for plotting.

The `trame_gui.py` module

Summary

Classes

TrameVisualizer	Defines the trame layout view.

TrameVisualizer

class TrameVisualizer

Defines the trame layout view.

Overview

Methods

set_scene	Set the trame layout view and the mesh to show through the PyVista plotter.
show	Start the trame server and show the mesh.

Import detail

```
from ansys.geometry.core.plotting.trame_gui import TrameVisualizer
```

Method detail

`TrameVisualizer.set_scene(plotter)`

Set the trame layout view and the mesh to show through the PyVista plotter.

Parameters

plotter
[`Plotter`] PyVista plotter with the rendered mesh.

`TrameVisualizer.show()`

Start the trame server and show the mesh.

Description

Module for using trame for visualization.

Description

Provides the PyAnsys Geometry plotting subpackage.

The `primitives` package

Summary

Submodules

circle	Provides for creating and managing a circle.
cone	Provides for creating and managing a cone.
curve_evaluation	Provides for creating and managing a curve.
cylinder	Provides for creating and managing a cylinder.
ellipse	Provides for creating and managing an ellipse.
line	Provides for creating and managing a line.
parameterization	Provides the parametrization-related classes.
sphere	Provides for creating and managing a sphere.
surface_evaluation	Provides for evaluating a surface.
torus	Provides for creating and managing a torus.

The `circle.py` module

Summary

Classes

Circle	Provides 3D circle representation.
CircleEvaluation	Provides evaluation of a circle at a given parameter.

Circle

class Circle(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*, *radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *reference: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X*, *axis: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Z*)

Provides 3D circle representation.

Overview

Methods

evaluate	Evaluate the circle at a given parameter.
transformed_copy	Create a transformed copy of the circle based on a transformation matrix.
mirrored_copy	Create a mirrored copy of the circle along the y-axis.
project_point	Project a point onto the circle and evauate the circle.
is_coincident_circle	Determine if the circle is coincident with another.
get_parameterization	Get the parametrization of the circle.

Properties

origin	Origin of the circle.
radius	Radius of the circle.
diameter	Diameter of the circle.
perimeter	Perimeter of the circle.
area	Area of the circle.
dir_x	X-direction of the circle.
dir_y	Y-direction of the circle.
dir_z	Z-direction of the circle.

Special methods

__eq__	Equals operator for the `Circle` class.

Import detail

```
from ansys.geometry.core.primitives.circle import Circle
```

Property detail

`property Circle.`**`origin`**`:` ***`Point3D`***

Origin of the circle.

`property Circle.`**`radius`**`: pint.Quantity`

Radius of the circle.

`property Circle.`**`diameter`**`: pint.Quantity`

Diameter of the circle.

`property Circle.`**`perimeter`**`: pint.Quantity`

Perimeter of the circle.

`property Circle.`**`area`**`: pint.Quantity`

Area of the circle.

`property Circle.`**`dir_x`**`:` ***`UnitVector3D`***

X-direction of the circle.

`property Circle.`**`dir_y`**`:` ***`UnitVector3D`***

Y-direction of the circle.

`property Circle.`**`dir_z`**`:` ***`UnitVector3D`***

Z-direction of the circle.

Method detail

`Circle.`**`__eq__`**(*other:* Circle) → bool

Equals operator for the `Circle` class.

`Circle.`**`evaluate`**(*parameter: ansys.geometry.core.typing.Real*) → *CircleEvaluation*

Evaluate the circle at a given parameter.

Parameters

parameter
[`Real`] Parameter to evaluate the circle at.

Returns

`CircleEvaluation`
Resulting evaluation.

`Circle.`**`transformed_copy`**(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Circle*

Create a transformed copy of the circle based on a transformation matrix.

Parameters

matrix
[*`Matrix44`*] 4x4 transformation matrix to apply to the circle.

Returns

Circle
New circle that is the transformed copy of the original circle.

`Circle.`**`mirrored_copy`**`()` → *Circle*

Create a mirrored copy of the circle along the y-axis.

Returns

Circle
A new circle that is a mirrored copy of the original circle.

`Circle.`**`project_point`**`(`*point:* ansys.geometry.core.math.point.Point3D`)` → *CircleEvaluation*

Project a point onto the circle and evauate the circle.

Parameters

point
[*Point3D*] Point to project onto the circle.

Returns

CircleEvaluation
Resulting evaluation.

`Circle.`**`is_coincident_circle`**`(`*other:* Circle`)` → bool

Determine if the circle is coincident with another.

Parameters

other
[*Circle*] Circle to determine coincidence with.

Returns

bool
`True` if this circle is coincident with the other, `False` otherwise.

`Circle.`**`get_parameterization`**`()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization of the circle.

The parameter of a circle specifies the clockwise angle around the axis (right-hand corkscrew law), with a zero parameter at `dir_x` and a period of 2*pi.

Returns

Parameterization
Information about how the circle is parameterized.

CircleEvaluation

class CircleEvaluation`(`*circle:* Circle, *parameter: ansys.geometry.core.typing.Real*`)`

Bases: *ansys.geometry.core.primitives.curve_evaluation.CurveEvaluation*

Provides evaluation of a circle at a given parameter.

Overview

Methods

position	Position of the evaluation.
tangent	Tangent of the evaluation.
normal	Normal to the circle.
first_derivative	First derivative of the evaluation.
second_derivative	Second derivative of the evaluation.
curvature	Curvature of the circle.

Properties

circle	Circle being evaluated.
parameter	Parameter that the evaluation is based upon.

Import detail

```python
from ansys.geometry.core.primitives.circle import CircleEvaluation
```

Property detail

property CircleEvaluation.**circle: *Circle***

Circle being evaluated.

property CircleEvaluation.**parameter: Real**

Parameter that the evaluation is based upon.

Method detail

CircleEvaluation.**position**() → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

Point3D
Point that lies on the circle at this evaluation.

CircleEvaluation.**tangent**() → *ansys.geometry.core.math.vector.UnitVector3D*

Tangent of the evaluation.

Returns

UnitVector3D
Tangent unit vector to the circle at this evaluation.

`CircleEvaluation.`**`normal`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Normal to the circle.

Returns

`UnitVector3D`
Normal unit vector to the circle at this evaluation.

`CircleEvaluation.`**`first_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative of the evaluation.

The first derivative is in the direction of the tangent and has a magnitude equal to the velocity (rate of change of position) at that point.

Returns

`Vector3D`
First derivative of the evaluation.

`CircleEvaluation.`**`second_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative of the evaluation.

Returns

`Vector3D`
Second derivative of the evaluation.

`CircleEvaluation.`**`curvature`**`()` → ansys.geometry.core.typing.Real

Curvature of the circle.

Returns

`Real`
Curvature of the circle.

Description

Provides for creating and managing a circle.

The `cone.py` module

Summary

Classes

`Cone`	Provides 3D cone representation.
`ConeEvaluation`	Evaluate the cone at given parameters.

Cone

class Cone(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*, *radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *half_angle: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]*, *reference: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X*, *axis: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Z*)

Provides 3D cone representation.

Overview

Methods

transformed_copy	Create a transformed copy of the cone based on a transformation matrix.
mirrored_copy	Create a mirrored copy of the cone along the y-axis.
evaluate	Evaluate the cone at given parameters.
project_point	Project a point onto the cone and evaluate the cone.
get_u_parameterization	Get the parametrization conditions for the U parameter.
get_v_parameterization	Get the parametrization conditions for the V parameter.

Properties

origin	Origin of the cone.
radius	Radius of the cone.
half_angle	Half angle of the apex.
dir_x	X-direction of the cone.
dir_y	Y-direction of the cone.
dir_z	Z-direction of the cone.
height	Height of the cone.
surface_area	Surface area of the cone.
volume	Volume of the cone.
apex	Apex point of the cone.
apex_param	Apex parameter of the cone.

Special methods

__eq__	Equals operator for the Cone class.

Import detail

```
from ansys.geometry.core.primitives.cone import Cone
```

Property detail

property Cone.**origin**: *Point3D*

Origin of the cone.

property Cone.**radius**: pint.Quantity

Radius of the cone.

property Cone.**half_angle**: pint.Quantity

Half angle of the apex.

property Cone.**dir_x**: *UnitVector3D*

X-direction of the cone.

property Cone.**dir_y**: *UnitVector3D*

Y-direction of the cone.

property Cone.**dir_z**: *UnitVector3D*

Z-direction of the cone.

property Cone.**height**: pint.Quantity

Height of the cone.

property Cone.**surface_area**: pint.Quantity

Surface area of the cone.

property Cone.**volume**: pint.Quantity

Volume of the cone.

property Cone.**apex**: *Point3D*

Apex point of the cone.

property Cone.**apex_param**: **Real**

Apex parameter of the cone.

Method detail

`Cone.`**`transformed_copy`**`(`*matrix:* ansys.geometry.core.math.matrix.Matrix44`)` → *Cone*

Create a transformed copy of the cone based on a transformation matrix.

Parameters

matrix
[*`Matrix44`*] 4x4 transformation matrix to apply to the cone.

Returns

`Cone`
New cone that is the transformed copy of the original cone.

`Cone.`**`mirrored_copy`**`()` → *Cone*

Create a mirrored copy of the cone along the y-axis.

Returns

`Cone`
New cone that is a mirrored copy of the original cone.

`Cone.`**`__eq__`**`(`*other:* Cone`)` → bool

Equals operator for the `Cone` class.

`Cone.`**`evaluate`**`(`*parameter:* ansys.geometry.core.primitives.parameterization.ParamUV`)` → *ConeEvaluation*

Evaluate the cone at given parameters.

Parameters

parameter
[*`ParamUV`*] Parameters (u,v) to evaluate the cone at.

Returns

`ConeEvaluation`
Resulting evaluation.

`Cone.`**`project_point`**`(`*point:* ansys.geometry.core.math.point.Point3D`)` → *ConeEvaluation*

Project a point onto the cone and evaluate the cone.

Parameters

point
[*`Point3D`*] Point to project onto the cone.

Returns

`ConeEvaluation`
Resulting evaluation.

`Cone.`**`get_u_parameterization`**`()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions for the U parameter.

The U parameter specifies the clockwise angle around the axis (right-hand corkscrew law), with a zero parameter at `dir_x` and a period of 2*pi.

Returns

`Parameterization`
Information about how a cone's U parameter is parameterized.

Cone.**get_v_parameterization**() → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions for the V parameter.

The V parameter specifies the distance along the axis, with a zero parameter at the XY plane of the cone.

Returns

Parameterization

Information about how a cone's V parameter is parameterized.

ConeEvaluation

class ConeEvaluation(*cone:* Cone, *parameter:* ansys.geometry.core.primitives.parameterization.ParamUV)

Bases: *ansys.geometry.core.primitives.surface_evaluation.SurfaceEvaluation*

Evaluate the cone at given parameters.

Overview

Methods

position	Position of the evaluation.
normal	Normal to the surface.
u_derivative	First derivative with respect to the U parameter.
v_derivative	First derivative with respect to the V parameter.
uu_derivative	Second derivative with respect to the U parameter.
uv_derivative	Second derivative with respect to the U and V parameters.
vv_derivative	Second derivative with respect to the V parameter.
min_curvature	Minimum curvature of the cone.
min_curvature_direction	Minimum curvature direction.
max_curvature	Maximum curvature of the cone.
max_curvature_direction	Maximum curvature direction.

Properties

cone	Cone being evaluated.
parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.cone import ConeEvaluation
```

Property detail

property `ConeEvaluation.`**`cone`**`:` ***`Cone`***

Cone being evaluated.

property `ConeEvaluation.`**`parameter`**`:` ***`ParamUV`***

Parameter that the evaluation is based upon.

Method detail

`ConeEvaluation.`**`position`**`()` → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

`Point3D`
Point that lies on the cone at this evaluation.

`ConeEvaluation.`**`normal`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Normal to the surface.

Returns

`UnitVector3D`
Normal unit vector to the cone at this evaluation.

`ConeEvaluation.`**`u_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the U parameter.

Returns

`Vector3D`
First derivative with respect to the U parameter.

`ConeEvaluation.`**`v_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the V parameter.

Returns

`Vector3D`
First derivative with respect to the V parameter.

`ConeEvaluation.`**`uu_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U parameter.

Returns

`Vector3D`
Second derivative with respect to the U parameter.

`ConeEvaluation.`**`uv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U and V parameters.

Returns

`Vector3D`
Second derivative with respect to U and V parameters.

`ConeEvaluation.`**`vv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the V parameter.

Returns

`Vector3D`
Second derivative with respect to the V parameter.

`ConeEvaluation.`**`min_curvature`**`()` → ansys.geometry.core.typing.Real

Minimum curvature of the cone.

Returns

`Real`
Minimum curvature of the cone.

`ConeEvaluation.`**`min_curvature_direction`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Minimum curvature direction.

Returns

`UnitVector3D`
Minimum curvature direction.

`ConeEvaluation.`**`max_curvature`**`()` → ansys.geometry.core.typing.Real

Maximum curvature of the cone.

Returns

`Real`
Maximum curvature of the cone.

`ConeEvaluation.`**`max_curvature_direction`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Maximum curvature direction.

Returns

`UnitVector3D`
Maximum curvature direction.

Description

Provides for creating and managing a cone.

The `curve_evaluation.py` module

Summary

Classes

`CurveEvaluation`	Provides for evaluating a curve.

CurveEvaluation

class CurveEvaluation(*parameter: ansys.geometry.core.typing.Real = None*)

Provides for evaluating a curve.

Overview

Abstract methods

position	Position of the evaluation.
first_derivative	First derivative of the evaluation.
second_derivative	Second derivative of the evaluation.
curvature	Curvature of the evaluation.

Methods

is_set	Determine if the parameter for the evaluation has been set.

Properties

parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.curve_evaluation import CurveEvaluation
```

Property detail

property CurveEvaluation.**parameter: Real**

Abstractmethod

Parameter that the evaluation is based upon.

Method detail

`CurveEvaluation.`**`is_set`**`()` → bool

Determine if the parameter for the evaluation has been set.

`abstract` `CurveEvaluation.`**`position`**`()` → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

`abstract` `CurveEvaluation.`**`first_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative of the evaluation.

`abstract` `CurveEvaluation.`**`second_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative of the evaluation.

`abstract` `CurveEvaluation.`**`curvature`**`()` → ansys.geometry.core.typing.Real

Curvature of the evaluation.

Description

Provides for creating and managing a curve.

The `cylinder.py` module

Summary

Classes

Cylinder	Provides 3D cylinder representation.
CylinderEvaluation	Provides evaluation of a cylinder at given parameters.

Cylinder

`class Cylinder``(`*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*], radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real], reference: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X, axis: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Z*`)`

Provides 3D cylinder representation.

Overview

Methods

surface_area	Get the surface area of the cylinder.
volume	Get the volume of the cylinder.
transformed_copy	Create a transformed copy of the cylinder based on a transformation matrix.
mirrored_copy	Create a mirrored copy of the cylinder along the y-axis.
evaluate	Evaluate the cylinder at the given parameters.
project_point	Project a point onto the cylinder and evaluate the cylinder.
get_u_parameterization	Get the parametrization conditions for the U parameter.
get_v_parameterization	Get the parametrization conditions for the V parameter.

Properties

origin	Origin of the cylinder.
radius	Radius of the cylinder.
dir_x	X-direction of the cylinder.
dir_y	Y-direction of the cylinder.
dir_z	Z-direction of the cylinder.

Special methods

__eq__	Equals operator for the `Cylinder` class.

Import detail

```
from ansys.geometry.core.primitives.cylinder import Cylinder
```

Property detail

property Cylinder.**origin:** ***Point3D***

Origin of the cylinder.

property Cylinder.**radius:** pint.Quantity

Radius of the cylinder.

property Cylinder.**dir_x:** ***UnitVector3D***

X-direction of the cylinder.

property Cylinder.**dir_y:** ***UnitVector3D***

Y-direction of the cylinder.

property Cylinder.**dir_z:** ***UnitVector3D***

Z-direction of the cylinder.

Method detail

`Cylinder.`**`surface_area`**(*height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*) → pint.Quantity

Get the surface area of the cylinder.

Parameters

height
[`Union`[`Quantity`, *`Distance`*, `Real`]] Height to bound the cylinder at.

Returns

`Quantity`
Surface area of the temporarily bounded cylinder.

Notes

By nature, a cylinder is infinite. If you want to get the surface area, you must bound it by a height. Normally a cylinder surface is not closed (does not have “caps” on the ends). This method assumes that the cylinder is closed for the purpose of getting the surface area.

`Cylinder.`**`volume`**(*height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*) → pint.Quantity

Get the volume of the cylinder.

Parameters

height
[`Union`[`Quantity`, *`Distance`*, `Real`]] Height to bound the cylinder at.

Returns

`Quantity`
Volume of the temporarily bounded cylinder.

Notes

By nature, a cylinder is infinite. If you want to get the surface area, you must bound it by a height. Normally a cylinder surface is not closed (does not have “caps” on the ends). This method assumes that the cylinder is closed for the purpose of getting the surface area.

`Cylinder.`**`transformed_copy`**(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Cylinder*

Create a transformed copy of the cylinder based on a transformation matrix.

Parameters

matrix
[*`Matrix44`*] 4X4 transformation matrix to apply to the cylinder.

Returns

`Cylinder`
New cylinder that is the transformed copy of the original cylinder.

`Cylinder.`**`mirrored_copy`**`()` → *Cylinder*

Create a mirrored copy of the cylinder along the y-axis.

Returns

`Cylinder`
New cylinder that is a mirrored copy of the original cylinder.

`Cylinder.`**`__eq__`**`(`*other:* Cylinder`)` → bool

Equals operator for the `Cylinder` class.

`Cylinder.`**`evaluate`**`(`*parameter:* ansys.geometry.core.primitives.parameterization.ParamUV`)` → *CylinderEvaluation*

Evaluate the cylinder at the given parameters.

Parameters

parameter
[*`ParamUV`*] Parameters (u,v) to evaluate the cylinder at.

Returns

`CylinderEvaluation`
Resulting evaluation.

`Cylinder.`**`project_point`**`(`*point:* ansys.geometry.core.math.point.Point3D`)` → *CylinderEvaluation*

Project a point onto the cylinder and evaluate the cylinder.

Parameters

point
[*`Point3D`*] Point to project onto the cylinder.

Returns

`CylinderEvaluation`
Resulting evaluation.

`Cylinder.`**`get_u_parameterization`**`()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions for the U parameter.

The U parameter specifies the clockwise angle around the axis (right-hand corkscrew law), with a zero parameter at `dir_x` and a period of 2*pi.

Returns

`Parameterization`
Information about how the cylinder's U parameter is parameterized.

`Cylinder.`**`get_v_parameterization`**`()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions for the V parameter.

The V parameter specifies the distance along the axis, with a zero parameter at the XY plane of the cylinder.

Returns

`Parameterization`
Information about how the cylinders's V parameter is parameterized.

CylinderEvaluation

class CylinderEvaluation(*cylinder:* Cylinder, *parameter:* ansys.geometry.core.primitives.parameterization.ParamUV)

Bases: *ansys.geometry.core.primitives.surface_evaluation.SurfaceEvaluation*

Provides evaluation of a cylinder at given parameters.

Overview

Methods

position	Position of the evaluation.
normal	Normal to the surface.
u_derivative	First derivative with respect to the U parameter.
v_derivative	First derivative with respect to the V parameter.
uu_derivative	Second derivative with respect to the U parameter.
uv_derivative	Second derivative with respect to the U and V parameters.
vv_derivative	Second derivative with respect to the V parameter.
min_curvature	Minimum curvature of the cylinder.
min_curvature_direction	Minimum curvature direction.
max_curvature	Maximum curvature of the cylinder.
max_curvature_direction	Maximum curvature direction.

Properties

cylinder	Cylinder being evaluated.
parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.cylinder import CylinderEvaluation
```

Property detail

property CylinderEvaluation.**cylinder**: ***Cylinder***

Cylinder being evaluated.

property CylinderEvaluation.**parameter**: ***ParamUV***

Parameter that the evaluation is based upon.

Method detail

`CylinderEvaluation.`**`position`**`()` → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

Point3D
Point that lies on the cylinder at this evaluation.

`CylinderEvaluation.`**`normal`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Normal to the surface.

Returns

UnitVector3D
Normal unit vector to the cylinder at this evaluation.

`CylinderEvaluation.`**`u_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the U parameter.

Returns

Vector3D
First derivative with respect to the U parameter.

`CylinderEvaluation.`**`v_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the V parameter.

Returns

Vector3D
First derivative with respect to the V parameter.

`CylinderEvaluation.`**`uu_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U parameter.

Returns

Vector3D
Second derivative with respect to the U parameter.

`CylinderEvaluation.`**`uv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U and V parameters.

Returns

Vector3D
Second derivative with respect to the U and v parameters.

`CylinderEvaluation.`**`vv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the V parameter.

Returns

Vector3D
Second derivative with respect to the V parameter.

`CylinderEvaluation.`**`min_curvature`**`()` → ansys.geometry.core.typing.Real

Minimum curvature of the cylinder.

Returns

Real
Minimum curvature of the cylinder.

CylinderEvaluation.**min_curvature_direction**() → *ansys.geometry.core.math.vector.UnitVector3D*

Minimum curvature direction.

Returns

UnitVector3D
Mminimum curvature direction.

CylinderEvaluation.**max_curvature**() → ansys.geometry.core.typing.Real

Maximum curvature of the cylinder.

Returns

Real
Maximum curvature of the cylinder.

CylinderEvaluation.**max_curvature_direction**() → *ansys.geometry.core.math.vector.UnitVector3D*

Maximum curvature direction.

Returns

UnitVector3D
Maximum curvature direction.

Description

Provides for creating and managing a cylinder.

The `ellipse.py` module

Summary

Classes

Ellipse	Provides 3D ellipse representation.
EllipseEvaluation	Evaluate an ellipse at a given parameter.

Ellipse

class Ellipse(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*, *major_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *minor_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *reference: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X*, *axis: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Z*)

Provides 3D ellipse representation.

Overview

Methods

mirrored_copy	Create a mirrored copy of the ellipse along the y-axis.
evaluate	Evaluate the ellipse at the given parameter.
project_point	Project a point onto the ellipse and evaluate the ellipse.
is_coincident_ellipse	Determine if this ellipse is coincident with another.
transformed_copy	Create a transformed copy of the ellipse based on a transformation matrix.
get_parameterization	Get the parametrization of the ellipse.

Properties

origin	Origin of the ellipse.
major_radius	Major radius of the ellipse.
minor_radius	Minor radius of the ellipse.
dir_x	X-direction of the ellipse.
dir_y	Y-direction of the ellipse.
dir_z	Z-direction of the ellipse.
eccentricity	Eccentricity of the ellipse.
linear_eccentricity	Linear eccentricity of the ellipse.
semi_latus_rectum	Semi-latus rectum of the ellipse.
perimeter	Perimeter of the ellipse.
area	Area of the ellipse.

Special methods

__eq__	Equals operator for the `Ellipse` class.

Import detail

```
from ansys.geometry.core.primitives.ellipse import Ellipse
```

Property detail

property Ellipse.**origin**: ***Point3D***

Origin of the ellipse.

property Ellipse.**major_radius**: pint.Quantity

Major radius of the ellipse.

property `Ellipse.`**`minor_radius`**`: pint.Quantity`

Minor radius of the ellipse.

property `Ellipse.`**`dir_x`**`: `*`UnitVector3D`*

X-direction of the ellipse.

property `Ellipse.`**`dir_y`**`: `*`UnitVector3D`*

Y-direction of the ellipse.

property `Ellipse.`**`dir_z`**`: `*`UnitVector3D`*

Z-direction of the ellipse.

property `Ellipse.`**`eccentricity`**`: Real`

Eccentricity of the ellipse.

property `Ellipse.`**`linear_eccentricity`**`: pint.Quantity`

Linear eccentricity of the ellipse.

Notes

The linear eccentricity is the distance from the center to the focus.

property `Ellipse.`**`semi_latus_rectum`**`: pint.Quantity`

Semi-latus rectum of the ellipse.

property `Ellipse.`**`perimeter`**`: pint.Quantity`

Perimeter of the ellipse.

property `Ellipse.`**`area`**`: pint.Quantity`

Area of the ellipse.

Method detail

`Ellipse.`**`__eq__`**(*other:* Ellipse) → bool

Equals operator for the `Ellipse` class.

`Ellipse.`**`mirrored_copy`**() → *Ellipse*

Create a mirrored copy of the ellipse along the y-axis.

Returns

`Ellipse`
New ellipse that is a mirrored copy of the original ellipse.

`Ellipse.`**`evaluate`**(*parameter: ansys.geometry.core.typing.Real*) → *EllipseEvaluation*

Evaluate the ellipse at the given parameter.

Parameters

parameter
[`Real`] Parameter to evaluate the ellipse at.

Returns

`EllipseEvaluation`
Resulting evaluation.

`Ellipse.`**`project_point`**`(`*point:* ansys.geometry.core.math.point.Point3D`)` → *EllipseEvaluation*

Project a point onto the ellipse and evaluate the ellipse.

Parameters

point
[*`Point3D`*] Point to project onto the ellipse.

Returns

`EllipseEvaluation`
Resulting evaluation.

`Ellipse.`**`is_coincident_ellipse`**`(`*other:* Ellipse`)` → bool

Determine if this ellipse is coincident with another.

Parameters

other
[*`Ellipse`*] Ellipse to determine coincidence with.

Returns

bool
`True` if this ellipse is coincident with the other, `False` otherwise.

`Ellipse.`**`transformed_copy`**`(`*matrix:* ansys.geometry.core.math.matrix.Matrix44`)` → *Ellipse*

Create a transformed copy of the ellipse based on a transformation matrix.

Parameters

matrix
[*`Matrix44`*] 4x4 transformation matrix to apply to the ellipse.

Returns

`Ellipse`
New ellipse that is the transformed copy of the original ellipse.

`Ellipse.`**`get_parameterization`**`()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization of the ellipse.

The parameter of an ellipse specifies the clockwise angle around the axis (right-hand corkscrew law), with a zero parameter at `dir_x` and a period of 2*pi.

Returns

`Parameterization`
Information about how the ellipse is parameterized.

EllipseEvaluation

`class EllipseEvaluation``(`*ellipse:* Ellipse, *parameter: ansys.geometry.core.typing.Real*`)`

Bases: *`ansys.geometry.core.primitives.curve_evaluation.CurveEvaluation`*

Evaluate an ellipse at a given parameter.

Overview

Methods

position	Position of the evaluation.
tangent	Tangent of the evaluation.
normal	Normal of the evaluation.
first_derivative	Girst derivative of the evaluation.
second_derivative	Second derivative of the evaluation.
curvature	Curvature of the ellipse.

Properties

ellipse	Ellipse being evaluated.
parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.ellipse import EllipseEvaluation
```

Property detail

property EllipseEvaluation.**ellipse: *Ellipse***

Ellipse being evaluated.

property EllipseEvaluation.**parameter: Real**

Parameter that the evaluation is based upon.

Method detail

EllipseEvaluation.**position**() → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

Point3D
Point that lies on the ellipse at this evaluation.

EllipseEvaluation.**tangent**() → *ansys.geometry.core.math.vector.UnitVector3D*

Tangent of the evaluation.

Returns

UnitVector3D
Tangent unit vector to the ellipse at this evaluation.

`EllipseEvaluation.`**`normal`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Normal of the evaluation.

Returns

`UnitVector3D`
Normal unit vector to the ellipse at this evaluation.

`EllipseEvaluation.`**`first_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Girst derivative of the evaluation.

The first derivative is in the direction of the tangent and has a magnitude equal to the velocity (rate of change of position) at that point.

Returns

`Vector3D`
First derivative of the evaluation.

`EllipseEvaluation.`**`second_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative of the evaluation.

Returns

`Vector3D`
Second derivative of the evaluation.

`EllipseEvaluation.`**`curvature`**`()` → ansys.geometry.core.typing.Real

Curvature of the ellipse.

Returns

`Real`
Curvature of the ellipse.

Description

Provides for creating and managing an ellipse.

The `line.py` module

Summary

Classes

`Line`	Provides 3D line representation.
`LineEvaluation`	Evaluate a line.

Line

class Line(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*], direction: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*]*)

Provides 3D line representation.

Overview

Methods

evaluate	Evaluate the line at a given parameter.
transformed_copy	Create a transformed copy of the line based on a transformation matrix.
project_point	Project a point onto the line and evaluate the line.
is_coincident_line	Determine if the line is coincident with another line.
is_opposite_line	Determine if the line is opposite another line.
get_parameterization	Get the parametrization of the line.

Properties

origin	Origin of the line.
direction	Direction of the line.

Special methods

__eq__	Equals operator for the `Line` class.

Import detail

```
from ansys.geometry.core.primitives.line import Line
```

Property detail

property Line.**origin:** ***Point3D***

Origin of the line.

property Line.**direction:** ***UnitVector3D***

Direction of the line.

Method detail

`Line.`**`__eq__`**(*other: object*) → bool

Equals operator for the `Line` class.

`Line.`**`evaluate`**(*parameter: float*) → *LineEvaluation*

Evaluate the line at a given parameter.

Parameters

parameter
[`Real`] Parameter to evaluate the line at.

Returns

`LineEvaluation`
Resulting evaluation.

`Line.`**`transformed_copy`**(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Line*

Create a transformed copy of the line based on a transformation matrix.

Parameters

matrix
[*`Matrix44`*] 4X4 transformation matrix to apply to the line.

Returns

`Line`
New line that is the transformed copy of the original line.

`Line.`**`project_point`**(*point:* ansys.geometry.core.math.point.Point3D) → *LineEvaluation*

Project a point onto the line and evaluate the line.

Parameters

point
[*`Point3D`*] Point to project onto the line.

Returns

`LineEvaluation`
Resulting evaluation.

`Line.`**`is_coincident_line`**(*other:* Line) → bool

Determine if the line is coincident with another line.

Parameters

other
[*`Line`*] Line to determine coincidence with.

Returns

bool
`True` if the line is coincident with another line, `False` otherwise.

`Line.`**`is_opposite_line`**(*other:* Line) → bool

Determine if the line is opposite another line.

Parameters

other
[*`Line`*] Line to determine opposition with.

Returns

bool
: `True` if the line is opposite to another line.

`Line.`**`get_parameterization`**`()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization of the line.

The parameter of a line specifies the distance from the *origin* in the direction of *direction*.

Returns

`Parameterization`
: Information about how the line is parameterized.

LineEvaluation

`class LineEvaluation`(*line:* Line, *parameter: float* = *None*)

Bases: *`ansys.geometry.core.primitives.curve_evaluation.CurveEvaluation`*

Evaluate a line.

Overview

Methods

`position`	Position of the evaluation.
`tangent`	Tangent of the evaluation, which is always equal to the direction of the line.
`first_derivative`	First derivative of the evaluation.
`second_derivative`	Second derivative of the evaluation.
`curvature`	Curvature of the line, which is always `0`.

Properties

`line`	Line being evaluated.
`parameter`	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.line import LineEvaluation
```

Property detail

property `LineEvaluation.`**`line:`** ***`Line`***

Line being evaluated.

property `LineEvaluation.`**`parameter:`** `float`

Parameter that the evaluation is based upon.

Method detail

`LineEvaluation.`**`position`**`()` → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

`Point3D`
Point that lies on the line at this evaluation.

`LineEvaluation.`**`tangent`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Tangent of the evaluation, which is always equal to the direction of the line.

Returns

`UnitVector3D`
Tangent unit vector to the line at this evaluation.

`LineEvaluation.`**`first_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative of the evaluation.

The first derivative is always equal to the direction of the line.

Returns

`Vector3D`
First derivative of the evaluation.

`LineEvaluation.`**`second_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative of the evaluation.

The second derivative is always equal to a zero vector `Vector3D([0, 0, 0])`.

Returns

`Vector3D`
Second derivative of the evaluation, which is always `Vector3D([0, 0, 0])`.

`LineEvaluation.`**`curvature`**`()` → float

Curvature of the line, which is always `0`.

Returns

`Real`
Curvature of the line, which is always `0`.

Description

Provides for creating and managing a line.

The `parameterization.py` module

Summary

Classes

ParamUV	Parameter class containing 2 parameters: (u, v).
Interval	Interval class that defines a range of values.
Parameterization	Parameterization class describes the parameters of a specific geometry.

Enums

ParamForm	ParamForm enum class that defines the form of a Parameterization.
ParamType	ParamType enum class that defines the type of a Parameterization.

ParamUV

class ParamUV(*u: ansys.geometry.core.typing.Real*, *v: ansys.geometry.core.typing.Real*)

Parameter class containing 2 parameters: (u, v).

Overview

Properties

u	u-parameter.
v	v-parameter.

Special methods

__add__	Add the u and v components of the other ParamUV to this ParamUV.
__sub__	Subtract the u and v components of the other ParamUV from this ParamUV.
__mul__	Multiplies the u and v components of this ParamUV by the other ParamUV.
__truediv__	Divides the u and v components of this ParamUV by the other ParamUV.
__repr__	Represent the `ParamUV` as a string.

Import detail

```
from ansys.geometry.core.primitives.parameterization import ParamUV
```

Property detail

property `ParamUV.`**u: Real**

u-parameter.

property `ParamUV.`**v: Real**

v-parameter.

Method detail

`ParamUV.`**`__add__`**(*other:* ParamUV) → *ParamUV*

Add the u and v components of the other ParamUV to this ParamUV.

Parameters

other
[*ParamUV*] The parameters to add these parameters.

Returns

ParamUV
The sum of the parameters.

`ParamUV.`**`__sub__`**(*other:* ParamUV) → *ParamUV*

Subtract the u and v components of the other ParamUV from this ParamUV.

Parameters

other
[*ParamUV*] The parameters to subtract from these parameters.

Returns

ParamUV
The difference of the parameters.

`ParamUV.`**`__mul__`**(*other:* ParamUV) → *ParamUV*

Multiplies the u and v components of this ParamUV by the other ParamUV.

Parameters

other
[*ParamUV*] The parameters to multiply by these parameters.

Returns

ParamUV
The product of the parameters.

`ParamUV.`**`__truediv__`**(*other:* ParamUV) → *ParamUV*

Divides the u and v components of this ParamUV by the other ParamUV.

Parameters

other
[*ParamUV*] The parameters to divide these parameters by.

Returns

ParamUV
The quotient of the parameters.

`ParamUV.`**`__repr__`**`()` → str
Represent the `ParamUV` as a string.

Interval

`class Interval`(*start: ansys.geometry.core.typing.Real*, *end: ansys.geometry.core.typing.Real*)

Interval class that defines a range of values.

Overview

Methods

is_open	If the interval is open (-inf, inf).
is_closed	If the interval is closed. Neither value is inf or -inf.
get_span	Return the quantity contained by the interval. Interval must be closed.

Properties

start	Start value of the interval.
end	End value of the interval.

Special methods

__repr__	Represent the `Interval` as a string.

Import detail

```
from ansys.geometry.core.primitives.parameterization import Interval
```

Property detail

property `Interval.`**`start: Real`**

Start value of the interval.

property `Interval.`**`end: Real`**

End value of the interval.

Method detail

`Interval.`**`is_open`**`()` → bool

If the interval is open (-inf, inf).

Returns

bool
True if both ends of the interval are negative and positive infinity respectively.

`Interval.`**`is_closed`**`()` → bool

If the interval is closed. Neither value is inf or -inf.

Returns

bool
True if neither bound of the interval is infinite.

`Interval.`**`get_span`**`()` → ansys.geometry.core.typing.Real

Return the quantity contained by the interval. Interval must be closed.

Returns

`Real`
The difference between the end and start of the interval.

`Interval.`**`__repr__`**`()` → str

Represent the `Interval` as a string.

Parameterization

class Parameterization(*form:* ParamForm, *type:* ParamType, *interval:* Interval)

Parameterization class describes the parameters of a specific geometry.

Overview

Properties

`form`	The form of the parameterization.
`type`	The type of the parameterization.
`interval`	The interval of the parameterization.

Special methods

__repr__	Represent the `Parameterization` as a string.

Import detail

```
from ansys.geometry.core.primitives.parameterization import Parameterization
```

Property detail

property Parameterization.**form**: ***ParamForm***

The form of the parameterization.

property Parameterization.**type**: ***ParamType***

The type of the parameterization.

property Parameterization.**interval**: ***Interval***

The interval of the parameterization.

Method detail

Parameterization.**__repr__**() → str

Represent the `Parameterization` as a string.

ParamForm

class ParamForm

Bases: `enum.Enum`

ParamForm enum class that defines the form of a Parameterization.

Overview

Attributes

OPEN
CLOSED
PERIODIC
OTHER

Import detail

```
from ansys.geometry.core.primitives.parameterization import ParamForm
```

Attribute detail

ParamForm.**OPEN** = 1

ParamForm.**CLOSED** = 2

ParamForm.**PERIODIC** = 3

ParamForm.**OTHER** = 4

ParamType

class ParamType

Bases: `enum.Enum`

ParamType enum class that defines the type of a Parameterization.

Overview

Attributes

LINEAR
CIRCULAR
OTHER

Import detail

```
from ansys.geometry.core.primitives.parameterization import ParamType
```

Attribute detail

ParamType.**LINEAR** = 1

ParamType.**CIRCULAR** = 2

ParamType.**OTHER** = 3

Description

Provides the parametrization-related classes.

The `sphere.py` module

Summary

Classes

Sphere	Provides 3D sphere representation.
SphereEvaluation	Evaluate a sphere at given parameters.

Sphere

class Sphere(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*]*, *radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *reference: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X*, *axis: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D*,* ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Z*)

Provides 3D sphere representation.

Overview

Methods

transformed_copy	Create a transformed copy of the sphere based on a transformation matrix.
mirrored_copy	Create a mirrored copy of the sphere along the y-axis.
evaluate	Evaluate the sphere at the given parameters.
project_point	Project a point onto the sphere and evaluate the sphere.
get_u_parameterization	Get the parametrization conditions for the U parameter.
get_v_parameterization	Get the parametrization conditions for the V parameter.

Properties

origin	Origin of the sphere.
radius	Radius of the sphere.
dir_x	X-direction of the sphere.
dir_y	Y-direction of the sphere.
dir_z	Z-direction of the sphere.
surface_area	Surface area of the sphere.
volume	Volume of the sphere.

Special methods

`__eq__`	Equals operator for the `Sphere` class.

Import detail

```
from ansys.geometry.core.primitives.sphere import Sphere
```

Property detail

property Sphere.**origin:** ***Point3D***

Origin of the sphere.

property Sphere.**radius:** pint.Quantity

Radius of the sphere.

property Sphere.**dir_x:** ***UnitVector3D***

X-direction of the sphere.

property Sphere.**dir_y:** ***UnitVector3D***

Y-direction of the sphere.

property Sphere.**dir_z:** ***UnitVector3D***

Z-direction of the sphere.

property Sphere.**surface_area:** pint.Quantity

Surface area of the sphere.

property Sphere.**volume:** pint.Quantity

Volume of the sphere.

Method detail

Sphere.**__eq__**(*other:* Sphere) → bool

Equals operator for the `Sphere` class.

Sphere.**transformed_copy**(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Sphere*

Create a transformed copy of the sphere based on a transformation matrix.

Parameters

matrix
[*Matrix44*] 4X4 transformation matrix to apply to the sphere.

Returns

Sphere
New sphere that is the transformed copy of the original sphere.

`Sphere.mirrored_copy()` → *Sphere*

Create a mirrored copy of the sphere along the y-axis.

Returns

Sphere
New sphere that is a mirrored copy of the original sphere.

`Sphere.evaluate(`*parameter:* ansys.geometry.core.primitives.parameterization.ParamUV`)` → *SphereEvaluation*

Evaluate the sphere at the given parameters.

Parameters

parameter
[*ParamUV*] Parameters (u,v) to evaluate the sphere at.

Returns

SphereEvaluation
Resulting evaluation.

`Sphere.project_point(`*point:* ansys.geometry.core.math.point.Point3D`)` → *SphereEvaluation*

Project a point onto the sphere and evaluate the sphere.

Parameters

point
[*Point3D*] Point to project onto the sphere.

Returns

SphereEvaluation
Resulting evaluation.

`Sphere.get_u_parameterization()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions for the U parameter.

The U parameter specifies the longitude angle, increasing clockwise (east) about `dir_z` (right-hand corkscrew law). It has a zero parameter at `dir_x` and a period of `2*pi`.

Returns

Parameterization
Information about how a sphere's U parameter is parameterized.

`Sphere.get_v_parameterization()` → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions for the V parameter.

The V parameter specifies the latitude, increasing north, with a zero parameter at the equator and a range of `[-pi/2, pi/2]`.

Returns

Parameterization
Information about how a sphere's V parameter is parameterized.

SphereEvaluation

class SphereEvaluation(*sphere:* Sphere, *parameter:* ansys.geometry.core.primitives.parameterization.ParamUV)

Bases: *ansys.geometry.core.primitives.surface_evaluation.SurfaceEvaluation*

Evaluate a sphere at given parameters.

Overview

Methods

position	Position of the evaluation.
normal	The normal to the surface.
u_derivative	First derivative with respect to the U parameter.
v_derivative	First derivative with respect to the V parameter.
uu_derivative	Second derivative with respect to the U parameter.
uv_derivative	Second derivative with respect to the U and V parameters.
vv_derivative	Second derivative with respect to the V parameter.
min_curvature	Minimum curvature of the sphere.
min_curvature_direction	Minimum curvature direction.
max_curvature	Maximum curvature of the sphere.
max_curvature_direction	Maximum curvature direction.

Properties

sphere	Sphere being evaluated.
parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.sphere import SphereEvaluation
```

Property detail

property SphereEvaluation.**sphere:** ***Sphere***

Sphere being evaluated.

property SphereEvaluation.**parameter:** ***ParamUV***

Parameter that the evaluation is based upon.

Method detail

`SphereEvaluation.`**`position`**`()` → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

`Point3D`
Point that lies on the sphere at this evaluation.

`SphereEvaluation.`**`normal`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

The normal to the surface.

Returns

`UnitVector3D`
Normal unit vector to the sphere at this evaluation.

`SphereEvaluation.`**`u_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the U parameter.

Returns

`Vector3D`
First derivative with respect to the U parameter.

`SphereEvaluation.`**`v_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the V parameter.

Returns

`Vector3D`
First derivative with respect to the V parameter.

`SphereEvaluation.`**`uu_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U parameter.

Returns

`Vector3D`
Second derivative with respect to the U parameter.

`SphereEvaluation.`**`uv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U and V parameters.

Returns

`Vector3D`
The second derivative with respect to the U and V parameters.

`SphereEvaluation.`**`vv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the V parameter.

Returns

`Vector3D`
The second derivative with respect to the V parameter.

`SphereEvaluation.`**`min_curvature`**`()` → ansys.geometry.core.typing.Real

Minimum curvature of the sphere.

Returns

Real
Minimum curvature of the sphere.

`SphereEvaluation.`**`min_curvature_direction`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Minimum curvature direction.

Returns

`UnitVector3D`
Minimum curvature direction.

`SphereEvaluation.`**`max_curvature`**`()` → ansys.geometry.core.typing.Real

Maximum curvature of the sphere.

Returns

`Real`
Maximum curvature of the sphere.

`SphereEvaluation.`**`max_curvature_direction`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Maximum curvature direction.

Returns

`UnitVector3D`
Maximum curvature direction.

Description

Provides for creating and managing a sphere.

The `surface_evaluation.py` module

Summary

Classes

`SurfaceEvaluation`	Provides for evaluating a surface.

SurfaceEvaluation

`class SurfaceEvaluation``(`*parameter:* ansys.geometry.core.primitives.parameterization.ParamUV`)`

Provides for evaluating a surface.

Overview

Abstract methods

position	Point on the surface, based on the evaluation.
normal	Normal to the surface.
u_derivative	First derivative with respect to the U parameter.
v_derivative	First derivative with respect to the V parameter.
uu_derivative	Second derivative with respect to the U parameter.
uv_derivative	The second derivative with respect to the U and V parameters.
vv_derivative	The second derivative with respect to v.
min_curvature	Minimum curvature.
min_curvature_direction	Minimum curvature direction.
max_curvature	Maximum curvature.
max_curvature_direction	Maximum curvature direction.

Properties

parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.surface_evaluation import SurfaceEvaluation
```

Property detail

property SurfaceEvaluation.**parameter: Real**

Abstractmethod

Parameter that the evaluation is based upon.

Method detail

abstract SurfaceEvaluation.**position**() → *ansys.geometry.core.math.point.Point3D*

Point on the surface, based on the evaluation.

abstract SurfaceEvaluation.**normal**() → *ansys.geometry.core.math.vector.UnitVector3D*

Normal to the surface.

abstract SurfaceEvaluation.**u_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the U parameter.

abstract SurfaceEvaluation.**v_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the V parameter.

abstract SurfaceEvaluation.**uu_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U parameter.

abstract SurfaceEvaluation.**uv_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

The second derivative with respect to the U and V parameters.

abstract SurfaceEvaluation.**vv_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

The second derivative with respect to v.

abstract SurfaceEvaluation.**min_curvature**() → ansys.geometry.core.typing.Real

Minimum curvature.

abstract SurfaceEvaluation.**min_curvature_direction**() → *ansys.geometry.core.math.vector.UnitVector3D*

Minimum curvature direction.

abstract SurfaceEvaluation.**max_curvature**() → ansys.geometry.core.typing.Real

Maximum curvature.

abstract SurfaceEvaluation.**max_curvature_direction**() → *ansys.geometry.core.math.vector.UnitVector3D*

Maximum curvature direction.

Description

Provides for evaluating a surface.

The `torus.py` module

Summary

Classes

Torus	Provides 3D torus representation.
TorusEvaluation	Evaluate the torus`` at given parameters.

Torus

class Torus(*origin: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.point.Point3D*], major_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real], minor_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real], reference: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_X, axis: beartype.typing.Union[numpy.ndarray, ansys.geometry.core.typing.RealSequence,* ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D*] = UNITVECTOR3D_Z*)

Provides 3D torus representation.

Overview

Methods

transformed_copy	Create a transformed copy of the torus based on a transformation matrix.
mirrored_copy	Create a mirrored copy of the torus along the y-axis.
evaluate	Evaluate the torus at the given parameters.
get_u_parameterization	Get the parametrization conditions for the U parameter.
get_v_parameterization	Get the parametrization conditions of the V parameter.
project_point	Project a point onto the torus and evaluate the torus.

Properties

origin	Origin of the torus.
major_radius	Semi-major radius of the torus.
minor_radius	Semi-minor radius of the torus.
dir_x	X-direction of the torus.
dir_y	Y-direction of the torus.
dir_z	Z-direction of the torus.
volume	Volume of the torus.
surface_area	Surface_area of the torus.

Special methods

__eq__	Equals operator for the `Torus` class.

Import detail

```python
from ansys.geometry.core.primitives.torus import Torus
```

Property detail

property Torus.**origin**: ***Point3D***

Origin of the torus.

property Torus.**major_radius**: pint.Quantity

Semi-major radius of the torus.

property Torus.**minor_radius**: pint.Quantity

Semi-minor radius of the torus.

property Torus.**dir_x**: ***UnitVector3D***

X-direction of the torus.

property Torus.**dir_y**: *UnitVector3D*

Y-direction of the torus.

property Torus.**dir_z**: *UnitVector3D*

Z-direction of the torus.

property Torus.**volume**: pint.Quantity

Volume of the torus.

property Torus.**surface_area**: pint.Quantity

Surface_area of the torus.

Method detail

Torus.**__eq__**(*other:* Torus) → bool

Equals operator for the `Torus` class.

Torus.**transformed_copy**(*matrix:* ansys.geometry.core.math.matrix.Matrix44) → *Torus*

Create a transformed copy of the torus based on a transformation matrix.

Parameters

matrix
[*Matrix44*] 4x4 transformation matrix to apply to the torus.

Returns

Torus
New torus that is the transformed copy of the original torus.

Torus.**mirrored_copy**() → *Torus*

Create a mirrored copy of the torus along the y-axis.

Returns

Torus
New torus that is a mirrored copy of the original torus.

Torus.**evaluate**(*parameter:* ansys.geometry.core.primitives.parameterization.ParamUV) → *TorusEvaluation*

Evaluate the torus at the given parameters.

Parameters

parameter
[*ParamUV*] Parameters (u,v) to evaluate the torus at.

Returns

TorusEvaluation
Resulting evaluation.

Torus.**get_u_parameterization**()

Get the parametrization conditions for the U parameter.

The U parameter specifies the longitude angle, increasing clockwise (east) about the axis (right-hand corkscrew law). It has a zero parameter at `Geometry.Frame.DirX` and a period of `2*pi`.

Returns

Parameterization
Information about how a sphere's U parameter is parameterized.

Torus.**get_v_parameterization**() → *ansys.geometry.core.primitives.parameterization.Parameterization*

Get the parametrization conditions of the V parameter.

The V parameter specifies the latitude, increasing north, with a zero parameter at the equator. For the donut, where the `Geometry.Torus.MajorRadius` is greater than the `Geometry.Torus.MinorRadius`, the range is `[-pi, pi]` and the parameterization is periodic. For a degenerate torus, the range is restricted accordingly and the parameterization is non-periodic.

Returns

Parameterization
Information about how a torus's V parameter is parameterized.

Torus.**project_point**(*point:* ansys.geometry.core.math.point.Point3D) → *TorusEvaluation*

Project a point onto the torus and evaluate the torus.

Parameters

point
[*Point3D*] Point to project onto the torus.

Returns

TorusEvaluation
Resulting evaluation.

TorusEvaluation

class TorusEvaluation(*torus:* Torus, *parameter:* ansys.geometry.core.primitives.parameterization.ParamUV)

Bases: *ansys.geometry.core.primitives.surface_evaluation.SurfaceEvaluation*

Evaluate the torus`` at given parameters.

Overview

Methods

position	Position of the evaluation.
normal	Normal to the surface.
u_derivative	First derivative with respect to the U parameter.
v_derivative	First derivative with respect to the V parameter.
uu_derivative	Second derivative with respect to the U parameter.
uv_derivative	Second derivative with respect to the U and V parameters.
vv_derivative	Second derivative with respect to the V parameter.
curvature	Curvature of the torus.
min_curvature	Minimum curvature of the torus.
min_curvature_direction	Minimum curvature direction.
max_curvature	Maximum curvature of the torus.
max_curvature_direction	Maximum curvature direction.

Properties

torus	Torus being evaluated.
parameter	Parameter that the evaluation is based upon.

Import detail

```
from ansys.geometry.core.primitives.torus import TorusEvaluation
```

Property detail

property TorusEvaluation.**torus**: ***Torus***

Torus being evaluated.

property TorusEvaluation.**parameter**: ***ParamUV***

Parameter that the evaluation is based upon.

Method detail

TorusEvaluation.**position**() → *ansys.geometry.core.math.point.Point3D*

Position of the evaluation.

Returns

Point3D
Point that lies on the torus at this evaluation.

TorusEvaluation.**normal**() → *ansys.geometry.core.math.vector.UnitVector3D*

Normal to the surface.

Returns

UnitVector3D
Normal unit vector to the torus at this evaluation.

TorusEvaluation.**u_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the U parameter.

Returns

Vector3D
First derivative with respect to the U parameter.

TorusEvaluation.**v_derivative**() → *ansys.geometry.core.math.vector.Vector3D*

First derivative with respect to the V parameter.

Returns

Vector3D
First derivative with respect to the V parameter.

`TorusEvaluation.`**`uu_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U parameter.

Returns

`Vector3D`
Second derivative with respect to the U parameter.

`TorusEvaluation.`**`uv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the U and V parameters.

Returns

`Vector3D`
Second derivative with respect to the U and V parameters.

`TorusEvaluation.`**`vv_derivative`**`()` → *ansys.geometry.core.math.vector.Vector3D*

Second derivative with respect to the V parameter.

Returns

`Vector3D`
Second derivative with respect to the V parameter.

`TorusEvaluation.`**`curvature`**`()` → Tuple[ansys.geometry.core.typing.Real, *ansys.geometry.core.math.vector.Vector3D*, ansys.geometry.core.typing.Real, *ansys.geometry.core.math.vector.Vector3D*]

Curvature of the torus.

Returns

`Tuple[Real,` ***`Vector3D`***`, Real,` ***`Vector3D`***`]`
Minimum and maximum curvature value and direction, respectively.

`TorusEvaluation.`**`min_curvature`**`()` → ansys.geometry.core.typing.Real

Minimum curvature of the torus.

Returns

`Real`
Minimum curvature of the torus.

`TorusEvaluation.`**`min_curvature_direction`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Minimum curvature direction.

Returns

`UnitVector3D`
Minimum curvature direction.

`TorusEvaluation.`**`max_curvature`**`()` → ansys.geometry.core.typing.Real

Maximum curvature of the torus.

Returns

`Real`
Maximum curvature of the torus.

`TorusEvaluation.`**`max_curvature_direction`**`()` → *ansys.geometry.core.math.vector.UnitVector3D*

Maximum curvature direction.

Returns

UnitVector3D
Maximum curvature direction.

Description

Provides for creating and managing a torus.

Description

PyAnsys Geometry primitives subpackage.

The `sketch` package

Summary

Submodules

arc	Provides for creating and managing an arc.
box	Provides for creating and managing a box (quadrilateral).
circle	Provides for creating and managing a circle.
edge	Provides for creating and managing an edge.
ellipse	Provides for creating and managing an ellipse.
face	Provides for creating and managing a face (closed 2D sketch).
gears	Module for creating and managing gears.
polygon	Provides for creating and managing a polygon.
segment	Provides for creating and managing a segment.
sketch	Provides for creating and managing a sketch.
slot	Provides for creating and managing a slot.
trapezoid	Provides for creating and managing a trapezoid.
triangle	Provides for creating and managing a triangle.

The `arc.py` module

Summary

Classes

Arc	Provides for modeling an arc.

Arc

class Arc(*center:* ansys.geometry.core.math.point.Point2D, *start:* ansys.geometry.core.math.point.Point2D, *end:* ansys.geometry.core.math.point.Point2D, *clockwise: beartype.typing.Optional[bool] = False*)

Bases: `ansys.geometry.core.sketch.edge.SketchEdge`

Provides for modeling an arc.

Overview

Constructors

`from_three_points`	Create an arc from three given points.

Properties

`start`	Starting point of the arc line.
`end`	Ending point of the arc line.
`length`	Length of the arc.
`radius`	Radius of the arc.
`center`	Center point of the arc.
`angle`	Angle of the arc.
`is_clockwise`	Flag indicating whether the rotation of the angle is clockwise.
`sector_area`	Area of the sector of the arc.
`visualization_polydata`	VTK polydata representation for PyVista visualization.

Special methods

`__eq__`	Equals operator for the `Arc` class.
`__ne__`	Not equals operator for the `Arc` class.

Import detail

```
from ansys.geometry.core.sketch.arc import Arc
```

Property detail

property `Arc.`**`start`**: *`Point2D`*

Starting point of the arc line.

property `Arc.`**`end`**: *`Point2D`*

Ending point of the arc line.

property `Arc.`**`length`**: `pint.Quantity`

Length of the arc.

property `Arc.`**`radius`**: `pint.Quantity`

Radius of the arc.

property `Arc.`**`center`**: *`Point2D`*

Center point of the arc.

property `Arc.`**`angle`**: `pint.Quantity`

Angle of the arc.

property `Arc.`**`is_clockwise`**: `bool`

Flag indicating whether the rotation of the angle is clockwise.

Returns

bool
`True` if the sense of rotation is clockwise. `False` if the sense of rotation is counter-clockwise.

property `Arc.`**`sector_area`**: `pint.Quantity`

Area of the sector of the arc.

property `Arc.`**`visualization_polydata`**: `pyvista.PolyData`

VTK polydata representation for PyVista visualization.

Returns

`pyvista.PolyData`
VTK pyvista.Polydata configuration.

Notes

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Method detail

`Arc.`**`__eq__`**(*other:* Arc) → bool

Equals operator for the `Arc` class.

`Arc.`**`__ne__`**(*other:* Arc) → bool

Not equals operator for the `Arc` class.

classmethod `Arc.`**`from_three_points`**(*start:* ansys.geometry.core.math.point.Point2D, *inter:* ansys.geometry.core.math.point.Point2D, *end:* ansys.geometry.core.math.point.Point2D)

Create an arc from three given points.

Parameters

start
[*Point2D*] Starting point of the arc.

inter
[*Point2D*] Intermediate point (location) of the arc.

end
[*Point2D*] Ending point of the arc.

Returns

Arc
Arc generated from the three points.

Description

Provides for creating and managing an arc.

The `box.py` module

Summary

Classes

Box	Provides for modeling a box.

Box

class Box(*center:* ansys.geometry.core.math.point.Point2D, *width: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance, *ansys.geometry.core.typing.Real]*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*)

Bases: *ansys.geometry.core.sketch.face.SketchFace*

Provides for modeling a box.

Overview

Properties

center	Center point of the box.
width	Width of the box.
height	Height of the box.
perimeter	Perimeter of the box.
area	Area of the box.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.box import Box
```

Property detail

property Box.**center**: *Point2D*

Center point of the box.

property Box.**width**: pint.Quantity

Width of the box.

property Box.**height**: pint.Quantity

Height of the box.

property Box.**perimeter**: pint.Quantity

Perimeter of the box.

property Box.**area**: pint.Quantity

Area of the box.

property Box.**visualization_polydata**: pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Description

Provides for creating and managing a box (quadrilateral).

The `circle.py` module

Summary

Classes

SketchCircle	Provides for modeling a circle.

SketchCircle

class SketchCircle(*center:* ansys.geometry.core.math.point.Point2D, *radius:* *beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*,* *ansys.geometry.core.typing.Real]*, *plane:* ansys.geometry.core.math.plane.Plane = *Plane()*)

Bases: *ansys.geometry.core.sketch.face.SketchFace*, *ansys.geometry.core.primitives.circle.Circle*

Provides for modeling a circle.

Overview

Methods

plane_change	Redefine the plane containing the `SketchCircle` objects.

Properties

center	Center of the circle.
perimeter	Perimeter of the circle.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.circle import SketchCircle
```

Property detail

property SketchCircle.**center**: ***Point2D***

Center of the circle.

property SketchCircle.**perimeter**: pint.Quantity

Perimeter of the circle.

Notes

This property resolves the dilemma between using the `SkethFace.perimeter` property and the `Circle.perimeter` property.

property SketchCircle.**visualization_polydata**: pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

`pyvista.PolyData`
VTK pyvista.Polydata configuration.

Method detail

`SketchCircle.`**`plane_change`**`(`*plane:* ansys.geometry.core.math.plane.Plane`)` → None

Redefine the plane containing the `SketchCircle` objects.

Parameters

plane
[*`Plane`*] Desired new plane that is to contain the sketched circle.

Notes

This implies that their 3D definition might suffer changes.

Description

Provides for creating and managing a circle.

The `edge.py` module

Summary

Classes

`SketchEdge`	Provides for modeling edges forming sketched shapes.

SketchEdge

`class SketchEdge`

Provides for modeling edges forming sketched shapes.

Overview

Methods

`plane_change`	Redefine the plane containing `SketchEdge` objects.

Properties

start	Starting point of the edge.
end	Ending point of the edge.
length	Length of the edge.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.edge import SketchEdge
```

Property detail

property SketchEdge.**start**: ***Point2D***

Abstractmethod

Starting point of the edge.

property SketchEdge.**end**: ***Point2D***

Abstractmethod

Ending point of the edge.

property SketchEdge.**length**: pint.Quantity

Abstractmethod

Length of the edge.

property SketchEdge.**visualization_polydata**: pyvista.PolyData

Abstractmethod

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Method detail

SketchEdge.**plane_change**(*plane:* ansys.geometry.core.math.plane.Plane) → None

Redefine the plane containing `SketchEdge` objects.

Parameters

plane
[*Plane*] Desired new plane that is to contain the sketched edge.

Notes

This implies that their 3D definition might suffer changes. By default, this metho does nothing. It is required to be implemented in child `SketchEdge` classes.

Description

Provides for creating and managing an edge.

The `ellipse.py` module

Summary

Classes

SketchEllipse	Provides for modeling an ellipse.

SketchEllipse

class SketchEllipse(*center:* ansys.geometry.core.math.point.Point2D, *major_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *minor_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*, *plane:* ansys.geometry.core.math.plane.Plane = *Plane()*)

Bases: *ansys.geometry.core.sketch.face.SketchFace*, *ansys.geometry.core.primitives.ellipse.Ellipse*

Provides for modeling an ellipse.

Overview

Methods

plane_change	Redefine the plane containing `SketchEllipse` objects.

Properties

center	Center point of the ellipse.
angle	Orientation angle of the ellipse.
perimeter	Perimeter of the circle.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.ellipse import SketchEllipse
```

Property detail

property SketchEllipse.**center**: ***Point2D***

Center point of the ellipse.

property SketchEllipse.**angle**: pint.Quantity

Orientation angle of the ellipse.

property SketchEllipse.**perimeter**: pint.Quantity

Perimeter of the circle.

Notes

This property resolves the dilemma between using the `SkethFace.perimeter` property and the `Ellipse.perimeter` property.

property SketchEllipse.**visualization_polydata**: pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Method detail

SketchEllipse.**plane_change**(*plane:* ansys.geometry.core.math.plane.Plane) → None

Redefine the plane containing `SketchEllipse` objects.

Parameters

plane
[*Plane*] Desired new plane that is to contain the sketched ellipse.

Notes

This implies that their 3D definition might suffer changes.

Description

Provides for creating and managing an ellipse.

The `face.py` module

Summary

Classes

SketchFace	Provides for modeling a face.

SketchFace

`class SketchFace`

Provides for modeling a face.

Overview

Methods

plane_change	Redefine the plane containing `SketchFace` objects.

Properties

edges	List of all component edges forming the face.
perimeter	Perimeter of the face.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.face import SketchFace
```

Property detail

property `SketchFace.`**`edges`**`: beartype.typing.List[`*`ansys.geometry.core.sketch.edge.SketchEdge`*`]`

List of all component edges forming the face.

property `SketchFace.`**`perimeter`**`: pint.Quantity`

Perimeter of the face.

property `SketchFace.`**`visualization_polydata`**`: pyvista.PolyData`

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

`pyvista.PolyData`
VTK pyvista.Polydata configuration.

Method detail

`SketchFace.`**`plane_change`**`(`*plane:* ansys.geometry.core.math.plane.Plane`)` → None

Redefine the plane containing `SketchFace` objects.

Parameters

plane
[*`Plane`*] Desired new plane that is to contain the sketched face.

Notes

This implies that their 3D definition might suffer changes. This method does nothing by default. It is required to be implemented in child `SketchFace` classes.

Description

Provides for creating and managing a face (closed 2D sketch).

The `gears.py` module

Summary

Classes

`Gear`	Provides the base class for sketching gears.
`DummyGear`	Provides the dummy class for sketching gears.
`SpurGear`	Provides the class for sketching spur gears.

Gear

class Gear

Bases: *ansys.geometry.core.sketch.face.SketchFace*

Provides the base class for sketching gears.

Overview

Properties

visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.gears import Gear
```

Property detail

property Gear.**visualization_polydata**: pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

DummyGear

class DummyGear(*origin:* ansys.geometry.core.math.point.Point2D, *outer_radius: beartype.typing.Union[pint.Quantity*, ansys.geometry.core.misc.measurements.Distance, *ansys.geometry.core.typing.Real]*, *inner_radius: beartype.typing.Union[pint.Quantity*, ansys.geometry.core.misc.measurements.Distance, *ansys.geometry.core.typing.Real]*, *n_teeth: int*)

Bases: *Gear*

Provides the dummy class for sketching gears.

Import detail

```
from ansys.geometry.core.sketch.gears import DummyGear
```

SpurGear

class SpurGear(*origin:* ansys.geometry.core.math.point.Point2D, *module: ansys.geometry.core.typing.Real*, *pressure_angle: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]*, *n_teeth: int*)

Bases: *Gear*

Provides the class for sketching spur gears.

Overview

Properties

origin	Origin of the spur gear.
module	Module of the spur gear.
pressure_angle	Pressure angle of the spur gear.
n_teeth	Number of teeth of the spur gear.
ref_diameter	Reference diameter of the spur gear.
base_diameter	Base diameter of the spur gear.
addendum	Addendum of the spur gear.
dedendum	Dedendum of the spur gear.
tip_diameter	Tip diameter of the spur gear.
root_diameter	Root diameter of the spur gear.

Import detail

```
from ansys.geometry.core.sketch.gears import SpurGear
```

Property detail

property SpurGear.**origin:** ***Point2D***

Origin of the spur gear.

property SpurGear.**module: Real**

Module of the spur gear.

property SpurGear.**pressure_angle:** pint.Quantity

Pressure angle of the spur gear.

property SpurGear.**n_teeth:** int

Number of teeth of the spur gear.

property SpurGear.**ref_diameter: Real**

Reference diameter of the spur gear.

property SpurGear.**base_diameter: Real**

Base diameter of the spur gear.

property SpurGear.**addendum: Real**

Addendum of the spur gear.

property SpurGear.**dedendum: Real**

Dedendum of the spur gear.

property SpurGear.**tip_diameter: Real**

Tip diameter of the spur gear.

property SpurGear.**root_diameter: Real**

Root diameter of the spur gear.

Description

Module for creating and managing gears.

The `polygon.py` module

Summary

Classes

Polygon	Provides for modeling regular polygons.

Polygon

class Polygon(*center:* ansys.geometry.core.math.point.Point2D, *inner_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real], sides:* *int*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*)

Bases: `ansys.geometry.core.sketch.face.SketchFace`

Provides for modeling regular polygons.

Overview

Properties

center	Center point of the polygon.
inner_radius	Inner radius (apothem) of the polygon.
n_sides	Number of sides of the polygon.
angle	Orientation angle of the polygon.
length	Side length of the polygon.
outer_radius	Outer radius of the polygon.
perimeter	Perimeter of the polygon.
area	Area of the polygon.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.polygon import Polygon
```

Property detail

property Polygon.**center**: ***Point2D***

Center point of the polygon.

property Polygon.**inner_radius**: pint.Quantity

Inner radius (apothem) of the polygon.

property Polygon.**n_sides**: int

Number of sides of the polygon.

property Polygon.**angle**: pint.Quantity

Orientation angle of the polygon.

property Polygon.**length**: pint.Quantity

Side length of the polygon.

property Polygon.**outer_radius**: pint.Quantity

Outer radius of the polygon.

property Polygon.**perimeter**: pint.Quantity

Perimeter of the polygon.

property Polygon.**area**: pint.Quantity

Area of the polygon.

property Polygon.**visualization_polydata**: pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Description

Provides for creating and managing a polygon.

The `segment.py` module

Summary

Classes

SketchSegment	Provides segment representation of a line.

SketchSegment

class SketchSegment(*start:* ansys.geometry.core.math.point.Point2D, *end:* ansys.geometry.core.math.point.Point2D, *plane:* ansys.geometry.core.math.plane.Plane = *Plane()*)

Bases: *ansys.geometry.core.sketch.edge.SketchEdge*, *ansys.geometry.core.primitives.line.Line*

Provides segment representation of a line.

Overview

Methods

plane_change	Redefine the plane containing `SketchSegment` objects.

Properties

start	Starting point of the segment.
end	Ending point of the segment.
length	Length of the segment.
visualization_polydata	VTK polydata representation for PyVista visualization.

Special methods

__eq__	Equals operator for the `SketchSegment` class.
__ne__	Not equals operator for the `SketchSegment` class.

Import detail

```
from ansys.geometry.core.sketch.segment import SketchSegment
```

Property detail

property SketchSegment.**start**: ***Point2D***

Starting point of the segment.

property SketchSegment.**end**: ***Point2D***

Ending point of the segment.

property SketchSegment.**length**: **pint.Quantity**

Length of the segment.

property SketchSegment.**visualization_polydata**: **pyvista.PolyData**

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

`pyvista.PolyData`
VTK pyvista.Polydata configuration.

Method detail

SketchSegment.**__eq__**(*other:* SketchSegment) → bool

Equals operator for the `SketchSegment` class.

SketchSegment.**__ne__**(*other:* SketchSegment) → bool

Not equals operator for the `SketchSegment` class.

SketchSegment.**plane_change**(*plane:* ansys.geometry.core.math.plane.Plane) → None

Redefine the plane containing `SketchSegment` objects.

Parameters

plane
[*Plane*] Desired new plane that is to contain the sketched segment.

Notes

This implies that their 3D definition might suffer changes.

Description

Provides for creating and managing a segment.

The `sketch.py` module

Summary

Classes

Sketch	Provides for building 2D sketch elements.

Attributes

SketchObject	Type used to refer to both `SketchEdge` and `SketchFace` as possible values.

Sketch

class Sketch(*plane: beartype.typing.Optional[*ansys.geometry.core.math.plane.Plane*] = Plane()*)

Provides for building 2D sketch elements.

Overview

Methods

translate_sketch_plane	Translate the origin location of the active sketch plane.
translate_sketch_plane_by_offset	Translate the origin location of the active sketch plane by offsets.
translate_sketch_plane_by_distan	Translate the origin location active sketch plane by distance.
get	Get a list of shapes with a given tag.
face	Add a sketch face to the sketch.
edge	Add a sketch edge to the sketch.
select	Add all objects that match provided tags to the current context.
segment	Add a segment sketch object to the sketch plane.
segment_to_point	Add a segment to the sketch plane starting from the previous edge end point.
segment_from_point_and_vector	Add a segment to the sketch starting from a given starting point.
segment_from_vector	Add a segment to the sketch starting from the end point of the previous edge.
arc	Add an arc to the sketch plane.
arc_to_point	Add an arc to the sketch starting from the end point of the previous edge.
arc_from_three_points	Add an arc to the sketch plane from three given points.
triangle	Add a triangle to the sketch using given vertex points.
trapezoid	Add a triangle to the sketch using given vertex points.
circle	Add a circle to the plane at a given center.
box	Create a box on the sketch.
slot	Create a slot on the sketch.
ellipse	Create an ellipse on the sketch.
polygon	Create a polygon on the sketch.
dummy_gear	Create a dummy gear on the sketch.
spur_gear	Create a spur gear on the sketch.
tag	Add a tag to the active selection of sketch objects.
plot	Plot all objects of the sketch to the scene.
plot_selection	Plot the current selection to the scene.
sketch_polydata	Get polydata configuration for all objects of the sketch to the scene.
sketch_polydata_faces	Get polydata configuration for all faces of the sketch to the scene.
sketch_polydata_edges	Get polydata configuration for all edges of the sketch to the scene.

Properties

plane	Sketch plane configuration.
edges	List of all independently sketched edges.
faces	List of all independently sketched faces.

Import detail

```
from ansys.geometry.core.sketch.sketch import Sketch
```

Property detail

property Sketch.**plane**: ***Plane***

Sketch plane configuration.

property Sketch.**edges**: **beartype.typing.List[*ansys.geometry.core.sketch.edge.SketchEdge*]**

List of all independently sketched edges.

Notes

Independently sketched edges are not assigned to a face. Face edges are not included in this list.

property Sketch.**faces**: **beartype.typing.List[*ansys.geometry.core.sketch.face.SketchFace*]**

List of all independently sketched faces.

Method detail

Sketch.**translate_sketch_plane**(*translation:* ansys.geometry.core.math.vector.Vector3D) → *Sketch*

Translate the origin location of the active sketch plane.

Parameters

translation
[*Vector3D*] Vector defining the translation. Meters is the expected unit.

Returns

Sketch
Revised sketch state ready for further sketch actions.

Sketch.**translate_sketch_plane_by_offset**(*x: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*] = Quantity(0, DEFAULT_UNITS.LENGTH)*, *y: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*] = Quantity(0, DEFAULT_UNITS.LENGTH)*, *z: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*] = Quantity(0, DEFAULT_UNITS.LENGTH)*) → *Sketch*

Translate the origin location of the active sketch plane by offsets.

Parameters

x
[Union[Quantity, *Distance*], default: Quantity(0, DEFAULT_UNITS.LENGTH)] Amount to translate the origin of the x-direction.

y
[Union[Quantity, *Distance*], default: Quantity(0, DEFAULT_UNITS.LENGTH)] Amount to translate the origin of the y-direction.

z
[Union[`Quantity`, *`Distance`*], default: `Quantity`(0, `DEFAULT_UNITS.LENGTH`)] Amount to translate the origin of the z-direction.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`translate_sketch_plane_by_distance`**(*direction:* ansys.geometry.core.math.vector.UnitVector3D, *distance: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*]*) → *Sketch*

Translate the origin location active sketch plane by distance.

Parameters

direction
[*`UnitVector3D`*] Direction to translate the origin.

distance
[Union[`Quantity`, *`Distance`*]] Distance to translate the origin.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`get`**(*tag: str*) → beartype.typing.List[SketchObject]

Get a list of shapes with a given tag.

Parameters

tag
[`str`] Tag to query against.

`Sketch.`**`face`**(*face:* ansys.geometry.core.sketch.face.SketchFace, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add a sketch face to the sketch.

Parameters

face
[*`SketchFace`*] Face to add.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`edge`**(*edge:* ansys.geometry.core.sketch.edge.SketchEdge, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add a sketch edge to the sketch.

Parameters

edge
[*`SketchEdge`*] Edge to add.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

`Sketch.`**`select`**`(`**tags: str*`)` → *Sketch*

Add all objects that match provided tags to the current context.

`Sketch.`**`segment`**`(`*start:* ansys.geometry.core.math.point.Point2D, *end:* ansys.geometry.core.math.point.Point2D, *tag: beartype.typing.Optional[str] = None*`)` → *Sketch*

Add a segment sketch object to the sketch plane.

Parameters

start
[*`Point2D`*] Starting point of the line segment.

end
[*`Point2D`*] Ending point of the line segment.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

`Sketch.`**`segment_to_point`**`(`*end:* ansys.geometry.core.math.point.Point2D, *tag: beartype.typing.Optional[str] = None*`)` → *Sketch*

Add a segment to the sketch plane starting from the previous edge end point.

Parameters

end
[*`Point2D`*] Ending point of the line segment.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

Notes

The starting point of the created edge is based upon the current context of the sketch, such as the end point of a previously added edge.

`Sketch.`**`segment_from_point_and_vector`**`(`*start:* ansys.geometry.core.math.point.Point2D, *vector:* ansys.geometry.core.math.vector.Vector2D, *tag: beartype.typing.Optional[str] = None*`)`

Add a segment to the sketch starting from a given starting point.

Parameters

start
[*`Point2D`*] Starting point of the line segment.

vector
[*Vector2D*] Vector defining the line segment. Vector magnitude determines the segment endpoint. Vector magnitude is assumed to be in the same unit as the starting point.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

Notes

Vector magnitude determines the segment endpoint. Vector magnitude is assumed to use the same unit as the starting point.

`Sketch.`**`segment_from_vector`**(*vector:* ansys.geometry.core.math.vector.Vector2D, *tag: beartype.typing.Optional[str] = None*)

Add a segment to the sketch starting from the end point of the previous edge.

Parameters

vector
[*Vector2D*] Vector defining the line segment.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

Notes

The starting point of the created edge is based upon the current context of the sketch, such as the end point of a previously added edge.

Vector magnitude determines the segment endpoint. Vector magnitude is assumed to use the same unit as the starting point in the previous context.

`Sketch.`**`arc`**(*start:* ansys.geometry.core.math.point.Point2D, *end:* ansys.geometry.core.math.point.Point2D, *center:* ansys.geometry.core.math.point.Point2D, *clockwise: beartype.typing.Optional[bool] = False*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add an arc to the sketch plane.

Parameters

start
[*Point2D*] Starting point of the arc.

end
[*Point2D*] Ending point of the arc.

center
[*Point2D*] Center point of the arc.

clockwise
[bool, default: `False`] Whether the arc spans the angle clockwise between the start and end points. When `False `` (default), the arc spans the angle counter-clockwise. When ``True`, the arc spans the angle clockwise.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

`Sketch.`**`arc_to_point`**(*end:* ansys.geometry.core.math.point.Point2D, *center:* ansys.geometry.core.math.point.Point2D, *clockwise: beartype.typing.Optional[bool] = False*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add an arc to the sketch starting from the end point of the previous edge.

Parameters

end
[*`Point2D`*] Ending point of the arc.

center
[*`Point2D`*] Center point of the arc.

clockwise
[bool, default: `False`] Whether the arc spans the angle clockwise between the start and end points. When `False` (default), the arc spans the angle counter-clockwise. When `True`, the arc spans the angle clockwise.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

Sketch
Revised sketch state ready for further sketch actions.

Notes

The starting point of the created edge is based upon the current context of the sketch, such as the end point of a previously added edge.

`Sketch.`**`arc_from_three_points`**(*start:* ansys.geometry.core.math.point.Point2D, *inter:* ansys.geometry.core.math.point.Point2D, *end:* ansys.geometry.core.math.point.Point2D, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add an arc to the sketch plane from three given points.

Parameters

start
[*`Point2D`*] Starting point of the arc.

inter
[*`Point2D`*] Intermediate point (location) of the arc.

end
[*`Point2D`*] End point of the arc.

tag
[`str`, default: `None`] User-defined label for identifying the edge.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`triangle`**(*point1:* ansys.geometry.core.math.point.Point2D, *point2:* ansys.geometry.core.math.point.Point2D, *point3:* ansys.geometry.core.math.point.Point2D, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add a triangle to the sketch using given vertex points.

Parameters

point1
[*`Point2D`*] Point that represents a vertex of the triangle.

point2
[*`Point2D`*] Point that represents a vertex of the triangle.

point3
[*`Point2D`*] Point that represents a vertex of the triangle.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`trapezoid`**(*width: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *slant_angle: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]*, *nonsymmetrical_slant_angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = None*, *center: beartype.typing.Optional[*ansys.geometry.core.math.point.Point2D*] = ZERO_POINT2D*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add a triangle to the sketch using given vertex points.

Parameters

width
[`Union`[`Quantity`, *`Distance`*, `Real`]] Width of the slot main body.

height
[`Union`[`Quantity`, *`Distance`*, `Real`]] Height of the slot.

slant_angle
[`Union`[`Quantity`, *`Angle`*, `Real`]] Angle for trapezoid generation.

nonsymmetrical_slant_angle
[`Union`[`Quantity`, *`Angle`*, `Real`], default: `None`] Asymmetrical slant angles on each side of the trapezoid. The default is `None`, in which case the trapezoid is symmetrical.

center
[*`Point2D`*, default: (0, 0)] Center point of the trapezoid.

angle
: [Optional[Union[Quantity, *Angle*, Real]], default: 0] Placement angle for orientation alignment.

tag
: [str, default: None] User-defined label for identifying the face.

Returns

Sketch
: Revised sketch state ready for further sketch actions.

`Sketch.circle`(*center:* ansys.geometry.core.math.point.Point2D, *radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Add a circle to the plane at a given center.

Parameters

center: Point2D
: Center point of the circle.

radius
: [Union[Quantity, *Distance*, Real]] Radius of the circle.

tag
: [str, default: None] User-defined label for identifying the face.

Returns

Sketch
: Revised sketch state ready for further sketch actions.

`Sketch.box`(*center:* ansys.geometry.core.math.point.Point2D, *width: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance, *ansys.geometry.core.typing.Real]*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Create a box on the sketch.

Parameters

center: Point2D
: Center point of the box.

width
: [Union[Quantity, *Distance*, Real]] Width of the box.

height
: [Union[Quantity, *Distance*, Real]] Height of the box.

angle
: [Union[Quantity, Real], default: 0] Placement angle for orientation alignment.

tag
: [str, default: None] User-defined label for identifying the face.

Returns

Sketch
: Revised sketch state ready for further sketch actions.

`Sketch.slot`(*center:* ansys.geometry.core.math.point.Point2D, *width: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Create a slot on the sketch.

Parameters

center: Point2D
Center point of the slot.

width
[`Union`[`Quantity`, *`Distance`*, `Real`]] Width of the slot.

height
[`Union`[`Quantity`, *`Distance`*, `Real`]] Height of the slot.

angle
[`Union`[`Quantity`, *`Angle`*, `Real`], default: 0] Placement angle for orientation alignment.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.ellipse`(*center:* ansys.geometry.core.math.point.Point2D, *major_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *minor_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Create an ellipse on the sketch.

Parameters

center: Point2D
Center point of the ellipse.

major_radius
[`Union`[`Quantity`, *`Distance`*, `Real`]] Semi-major axis of the ellipse.

minor_radius
[`Union`[`Quantity`, *`Distance`*, `Real`]] Semi-minor axis of the ellipse.

angle
[`Union`[`Quantity`, *`Angle`*, `Real`], default: 0] Placement angle for orientation alignment.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`polygon`**(*center:* ansys.geometry.core.math.point.Point2D, *inner_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *sides: int*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Create a polygon on the sketch.

Parameters

center: Point2D
Center point of the polygon.

inner_radius
[`Union`[`Quantity`, *`Distance`*, `Real`]] Inner radius (apothem) of the polygon.

sides
[`int`] Number of sides of the polygon.

angle
[`Union`[`Quantity`, *`Angle`*, `Real`], default: 0] Placement angle for orientation alignment.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`dummy_gear`**(*origin:* ansys.geometry.core.math.point.Point2D, *outer_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *inner_radius: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *n_teeth: int*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Create a dummy gear on the sketch.

Parameters

origin
[*`Point2D`*] Origin of the gear.

outer_radius
[`Union`[`Quantity`, *`Distance`*, `Real`]] Outer radius of the gear.

inner_radius
[`Union`[`Quantity`, *`Distance`*, `Real`]] Inner radius of the gear.

n_teeth
[`int`] Number of teeth of the gear.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`spur_gear`**(*origin:* ansys.geometry.core.math.point.Point2D, *module: ansys.geometry.core.typing.Real*, *pressure_angle: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]*, *n_teeth: int*, *tag: beartype.typing.Optional[str] = None*) → *Sketch*

Create a spur gear on the sketch.

Parameters

origin
[`Point2D`] Origin of the spur gear.

module
[`Real`] Module of the spur gear. This is also the ratio between the pitch circle diameter in millimeters and the number of teeth.

pressure_angle
[`Union`[`Quantity`, `Angle`, `Real`]] Pressure angle of the spur gear.

n_teeth
[`int`] Number of teeth of the spur gear.

tag
[`str`, default: `None`] User-defined label for identifying the face.

Returns

`Sketch`
Revised sketch state ready for further sketch actions.

`Sketch.`**`tag`**(*tag: str*) → None

Add a tag to the active selection of sketch objects.

Parameters

tag
[`str`] Tag to assign to the sketch objects.

`Sketch.`**`plot`**(*view_2d: beartype.typing.Optional[bool] = False*, *screenshot: beartype.typing.Optional[str] = None*, *use_trame: beartype.typing.Optional[bool] = None*, *selected_pd_objects: beartype.typing.List[pyvista.PolyData] = None*, ***plotting_options: beartype.typing.Optional[dict]*)

Plot all objects of the sketch to the scene.

Parameters

view_2d
[bool, default: `False`] Whether to represent the plot in a 2D format.

screenshot
[`str`, `optional`] Path for saving a screenshot of the image that is being represented.

use_trame
[bool, default: `None`] Whether to enables the use of trame. The default is `None`, in which case the `USE_TRAME` global setting is used.

****plotting_options**
[`dict`, `optional`] Keyword arguments for plotting. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`Sketch.`**`plot_selection`**(*view_2d: beartype.typing.Optional[bool] = False*, *screenshot: beartype.typing.Optional[str] = None*, *use_trame: beartype.typing.Optional[bool] = None*, ***plotting_options: beartype.typing.Optional[dict]*)

Plot the current selection to the scene.

Parameters

view_2d
[bool, default: `False`] Whether to represent the plot in a 2D format.

screenshot
[str, `optional`] Path for saving a screenshot of the image that is being represented.

use_trame
[bool, default: `None`] Whether to enables the use of trame. The default is `None`, in which case the `USE_TRAME` global setting is used.

****plotting_options**
[dict, `optional`] Keyword arguments for plotting. For allowable keyword arguments, see the `Plotter.add_mesh` method.

`Sketch.`**`sketch_polydata`**`()` → beartype.typing.List[pyvista.PolyData]

Get polydata configuration for all objects of the sketch to the scene.

Returns

`List[PolyData]`
List of the polydata configuration for all edges and faces in the sketch.

`Sketch.`**`sketch_polydata_faces`**`()` → beartype.typing.List[pyvista.PolyData]

Get polydata configuration for all faces of the sketch to the scene.

Returns

`List[PolyData]`
List of the polydata configuration for faces in the sketch.

`Sketch.`**`sketch_polydata_edges`**`()` → beartype.typing.List[pyvista.PolyData]

Get polydata configuration for all edges of the sketch to the scene.

Returns

`List[PolyData]`
List of the polydata configuration for edges in the sketch.

Description

Provides for creating and managing a sketch.

Module detail

`sketch.`**`SketchObject`**

Type used to refer to both `SketchEdge` and `SketchFace` as possible values.

The `slot.py` module

Summary

Classes

Slot	Provides for modeling a 2D slot.

Slot

class Slot(*center:* ansys.geometry.core.math.point.Point2D, *width: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real]*, *angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*)

Bases: *ansys.geometry.core.sketch.face.SketchFace*

Provides for modeling a 2D slot.

Overview

Properties

center	Center of the slot.
width	Width of the slot.
height	Height of the slot.
perimeter	Perimeter of the slot.
area	Area of the slot.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.slot import Slot
```

Property detail

property Slot.**center**: *Point2D*

Center of the slot.

property Slot.**width**: pint.Quantity

Width of the slot.

property Slot.**height**: pint.Quantity

Height of the slot.

property Slot.**perimeter:** pint.Quantity

Perimeter of the slot.

property Slot.**area:** pint.Quantity

Area of the slot.

property Slot.**visualization_polydata:** pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Description

Provides for creating and managing a slot.

The `trapezoid.py` module

Summary

Classes

Trapezoid	Provides for modeling a 2D trapezoid.

Trapezoid

class Trapezoid(*width: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real], height: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Distance*, ansys.geometry.core.typing.Real], slant_angle: beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real], nonsymmetrical_slant_angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = None, center: beartype.typing.Optional[*ansys.geometry.core.math.point.Point2D*] = ZERO_POINT2D, angle: beartype.typing.Optional[beartype.typing.Union[pint.Quantity,* ansys.geometry.core.misc.measurements.Angle*, ansys.geometry.core.typing.Real]] = 0*)

Bases: `ansys.geometry.core.sketch.face.SketchFace`

Provides for modeling a 2D trapezoid.

Overview

Properties

center	Center of the trapezoid.
width	Width of the trapezoid.
height	Height of the trapezoid.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.trapezoid import Trapezoid
```

Property detail

property Trapezoid.**center:** ***Point2D***

Center of the trapezoid.

property Trapezoid.**width:** pint.Quantity

Width of the trapezoid.

property Trapezoid.**height:** pint.Quantity

Height of the trapezoid.

property Trapezoid.**visualization_polydata:** pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Description

Provides for creating and managing a trapezoid.

The `triangle.py` module

Summary

Classes

Triangle	Provides for modeling 2D triangles.

Triangle

class Triangle(*point1:* ansys.geometry.core.math.point.Point2D, *point2:* ansys.geometry.core.math.point.Point2D, *point3:* ansys.geometry.core.math.point.Point2D)

Bases: *ansys.geometry.core.sketch.face.SketchFace*

Provides for modeling 2D triangles.

Overview

Properties

point1	Triangle vertex 1.
point2	Triangle vertex 2.
point3	Triangle vertex 3.
visualization_polydata	VTK polydata representation for PyVista visualization.

Import detail

```
from ansys.geometry.core.sketch.triangle import Triangle
```

Property detail

property Triangle.**point1:** ***Point2D***

Triangle vertex 1.

property Triangle.**point2:** ***Point2D***

Triangle vertex 2.

property Triangle.**point3:** ***Point2D***

Triangle vertex 3.

property Triangle.**visualization_polydata:** pyvista.PolyData

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

Returns

pyvista.PolyData
VTK pyvista.Polydata configuration.

Description

Provides for creating and managing a triangle.

Description

PyAnsys Geometry sketch subpackage.

The `tools` package

Summary

Submodules

problem_areas	The problem area definition.
repair_tool_message	Module for repair tool message.
repair_tools	Provides tools for repairing bodies.

The `problem_areas.py` module

Summary

Classes

ProblemArea	Represents problem areas.
DuplicateFaceProblemAreas	Provides duplicate face problem area definition.
MissingFaceProblemAreas	Provides missing face problem area definition.
InexactEdgeProblemAreas	Represents an inexact edge problem area with unique identifier and associated edges.
ExtraEdgeProblemAreas	Represents a extra edge problem area with unique identifier and associated edges.
SmallFaceProblemAreas	Represents a small face problem area with unique identifier and associated faces.
SplitEdgeProblemAreas	Represents a split edge problem area with unique identifier and associated edges.
StitchFaceProblemAreas	Represents a stitch face problem area with unique identifier and associated faces.

ProblemArea

class ProblemArea(*id: str*, *grpc_client: ansys.geometry.core.connection.GrpcClient*)

Represents problem areas.

Overview

Abstract methods

fix	Fix problem area.

Properties

id	The id of the problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import ProblemArea
```

Property detail

property ProblemArea.**id**: str

The id of the problem area.

Method detail

abstract ProblemArea.**fix**()

Fix problem area.

DuplicateFaceProblemAreas

class DuplicateFaceProblemAreas(*id: str, faces: beartype.typing.List[str], grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *ProblemArea*

Provides duplicate face problem area definition.

Overview

Methods

fix	Fix the problem area.

Properties

faces	The list of the problem area ids.

Import detail

```
from ansys.geometry.core.tools.problem_areas import DuplicateFaceProblemAreas
```

Property detail

property DuplicateFaceProblemAreas.**faces: beartype.typing.List[str]**

The list of the problem area ids.

This method returns the list of problem area ids with duplicate faces.

Method detail

DuplicateFaceProblemAreas.**fix**() → *ansys.geometry.core.tools.repair_tool_message.RepairToolMessage*

Fix the problem area.

Returns

message: ***RepairToolMessage***
Message containing created and/or modified bodies.

MissingFaceProblemAreas

class MissingFaceProblemAreas(*id: str*, *edges: beartype.typing.List[str]*, *grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *ProblemArea*

Provides missing face problem area definition.

Overview

Methods

fix	Fix the problem area.

Properties

edges	The list of the ids of the edges connected to this problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import MissingFaceProblemAreas
```

Property detail

property `MissingFaceProblemAreas.`**`edges: beartype.typing.List[str]`**

The list of the ids of the edges connected to this problem area.

Method detail

`MissingFaceProblemAreas.`**`fix`**`()` → *ansys.geometry.core.tools.repair_tool_message.RepairToolMessage*

Fix the problem area.

Returns

message: ***`RepairToolMessage`***

Message containing created and/or modified bodies.

InexactEdgeProblemAreas

class InexactEdgeProblemAreas(*id: str, edges: beartype.typing.List[str], grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *`ProblemArea`*

Represents an inexact edge problem area with unique identifier and associated edges.

Overview

Methods

fix	Fix the problem area.

Properties

edges	The list of the ids of the edges connected to this problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import InexactEdgeProblemAreas
```

Property detail

property `InexactEdgeProblemAreas.`**`edges: beartype.typing.List[str]`**

The list of the ids of the edges connected to this problem area.

Method detail

`InexactEdgeProblemAreas.`**`fix`**`()` → *ansys.geometry.core.tools.repair_tool_message.RepairToolMessage*

Fix the problem area.

Returns

message: ***RepairToolMessage***
Message containing created and/or modified bodies.

ExtraEdgeProblemAreas

class ExtraEdgeProblemAreas(*id: str, edges: beartype.typing.List[str], grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *ProblemArea*

Represents a extra edge problem area with unique identifier and associated edges.

Overview

Properties

edges	The list of the ids of the edges connected to this problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import ExtraEdgeProblemAreas
```

Property detail

property ExtraEdgeProblemAreas.**edges: beartype.typing.List[str]**

The list of the ids of the edges connected to this problem area.

SmallFaceProblemAreas

class SmallFaceProblemAreas(*id: str*, *faces: beartype.typing.List[str]*, *grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *ProblemArea*

Represents a small face problem area with unique identifier and associated faces.

Overview

Methods

fix	Fix the problem area.

Properties

faces	The list of the ids of the edges connected to this problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import SmallFaceProblemAreas
```

Property detail

property SmallFaceProblemAreas.**faces: beartype.typing.List[str]**

The list of the ids of the edges connected to this problem area.

Method detail

`SmallFaceProblemAreas.fix()` → *ansys.geometry.core.tools.repair_tool_message.RepairToolMessage*

Fix the problem area.

Returns

message: ***RepairToolMessage***
Message containing created and/or modified bodies.

SplitEdgeProblemAreas

class SplitEdgeProblemAreas(*id: str*, *edges: beartype.typing.List[str]*, *grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *ProblemArea*

Represents a split edge problem area with unique identifier and associated edges.

Overview

Methods

fix	Fix the problem area.

Properties

edges	The list of the ids of the edges connected to this problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import SplitEdgeProblemAreas
```

Property detail

property SplitEdgeProblemAreas.**edges: beartype.typing.List[str]**

The list of the ids of the edges connected to this problem area.

Method detail

`SplitEdgeProblemAreas.`**`fix`**`()` → *ansys.geometry.core.tools.repair_tool_message.RepairToolMessage*

Fix the problem area.

Returns

message: ***`RepairToolMessage`***
Message containing created and/or modified bodies.

StitchFaceProblemAreas

`class StitchFaceProblemAreas`(*id: str, faces: beartype.typing.List[str], grpc_client: ansys.geometry.core.connection.GrpcClient*)

Bases: *`ProblemArea`*

Represents a stitch face problem area with unique identifier and associated faces.

Overview

Methods

`fix`	Fix the problem area.

Properties

`faces`	The list of the ids of the faces connected to this problem area.

Import detail

```
from ansys.geometry.core.tools.problem_areas import StitchFaceProblemAreas
```

Property detail

`property` `StitchFaceProblemAreas.`**`faces: beartype.typing.List[str]`**

The list of the ids of the faces connected to this problem area.

Method detail

`StitchFaceProblemAreas.fix()` → *ansys.geometry.core.tools.repair_tool_message.RepairToolMessage*

Fix the problem area.

Returns

message: ***RepairToolMessage***
Message containing created and/or modified bodies.

Description

The problem area definition.

The `repair_tool_message.py` module

Summary

Classes

RepairToolMessage	Provides return message for the repair tool methods.

RepairToolMessage

class RepairToolMessage(*success: bool, created_bodies: beartype.typing.List[str], modified_bodies: beartype.typing.List[str]*)

Provides return message for the repair tool methods.

Overview

Properties

success	The success of the repair operation.
created_bodies	The list of the created bodies after the repair operation.
modified_bodies	The list of the modified bodies after the repair operation.

Import detail

```
from ansys.geometry.core.tools.repair_tool_message import RepairToolMessage
```

Property detail

property RepairToolMessage.**success: bool**

The success of the repair operation.

property RepairToolMessage.**created_bodies: beartype.typing.List[str]**

The list of the created bodies after the repair operation.

property RepairToolMessage.**modified_bodies: beartype.typing.List[str]**

The list of the modified bodies after the repair operation.

Description

Module for repair tool message.

The `repair_tools.py` module

Summary

Classes

RepairTools	Repair tools for PyAnsys Geometry.

RepairTools

class RepairTools(*grpc_client: ansys.geometry.core.connection.GrpcClient*)

Repair tools for PyAnsys Geometry.

Overview

Methods

find_split_edges	Find split edges in the given list of bodies.
find_extra_edges	Find the extra edges in the given list of bodies.
find_inexact_edges	Find inexact edges in the given list of bodies.
find_duplicate_faces	Find the duplicate face problem areas.
find_missing_faces	Find the missing faces.
find_small_faces	Find the small face problem areas.
find_stitch_faces	Return the list of stitch face problem areas.

Import detail

```
from ansys.geometry.core.tools.repair_tools import RepairTools
```

Method detail

RepairTools.**find_split_edges**(*ids: beartype.typing.List[str], angle: ansys.geometry.core.typing.Real = 0.0, length: ansys.geometry.core.typing.Real = 0.0*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.SplitEdgeProblemAreas*]

Find split edges in the given list of bodies.

This method finds the split edge problem areas and returns a list of split edge problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

angle
[Real] The maximum angle between edges.

length
[Real] The maximum length of the edges.

Returns

List[*SplitEdgeProblemAreas*]
List of objects representing split edge problem areas.

RepairTools.**find_extra_edges**(*ids: beartype.typing.List[str]*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.ExtraEdgeProblemAreas*]

Find the extra edges in the given list of bodies.

This method find the extra edge problem areas and returns a list of extra edge problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

Returns

List[ExtraEdgeProblemArea]
List of objects representing extra edge problem areas.

RepairTools.**find_inexact_edges**(*ids*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.InexactEdgeProblemAreas*]

Find inexact edges in the given list of bodies.

This method find the inexact edge problem areas and returns a list of inexact edge problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

Returns

List[InExactEdgeProblemArea]
List of objects representing inexact edge problem areas.

RepairTools.**find_duplicate_faces**(*ids*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.DuplicateFaceProblemAreas*

Find the duplicate face problem areas.

This method finds the duplicate face problem areas and returns a list of duplicate face problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

Returns

List[*DuplicateFaceProblemAreas*]
List of objects representing duplicate face problem areas.

RepairTools.**find_missing_faces**(*ids*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.MissingFaceProblemAreas*]

Find the missing faces.

This method find the missing face problem areas and returns a list of missing face problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

Returns

List[*MissingFaceProblemAreas*]
List of objects representing missing face problem areas.

RepairTools.**find_small_faces**(*ids*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.SmallFaceProblemAreas*]

Find the small face problem areas.

This method finds and returns a list of ids of small face problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

Returns

List[*SmallFaceProblemAreas*]
List of objects representing small face problem areas.

RepairTools.**find_stitch_faces**(*ids*) → beartype.typing.List[*ansys.geometry.core.tools.problem_areas.StitchFaceProblemAreas*]

Return the list of stitch face problem areas.

This method find the stitch face problem areas and returns a list of ids of stitch face problem areas objects.

Parameters

ids
[List[str]] Server-defined ID for the bodies.

Returns

List[*StitchFaceProblemAreas*]
List of objects representing stitch face problem areas.

Description

Provides tools for repairing bodies.

Description

PyAnsys Geometry tools subpackage.

The `errors.py` module

Summary

Exceptions

GeometryRuntimeError	Provides error message to raise when Geometry service passes a runtime error.
GeometryExitedError	Provides error message to raise when Geometry service has exited.

Functions

handler	Pass signal to the custom interrupt handler.
protect_grpc	Capture gRPC exceptions and raise a more succinct error message.

Constants

SIGINT_TRACKER

GeometryRuntimeError

exception GeometryRuntimeError

Bases: `RuntimeError`

Provides error message to raise when Geometry service passes a runtime error.

Import detail

```
from ansys.geometry.core.errors import GeometryRuntimeError
```

GeometryExitedError

exception GeometryExitedError(*msg='Geometry service has exited.'*)

Bases: `RuntimeError`

Provides error message to raise when Geometry service has exited.

Import detail

```
from ansys.geometry.core.errors import GeometryExitedError
```

Description

Provides PyAnsys Geometry-specific errors.

Module detail

`errors.`**handler**(*sig*, *frame*)

Pass signal to the custom interrupt handler.

`errors.`**protect_grpc**(*func*)

Capture gRPC exceptions and raise a more succinct error message.

This method captures the `KeyboardInterrupt` exception to avoid segfaulting the Geometry service.

While this works some of the time, it does not work all of the time. For some reason, gRPC still captures SIGINT.

`errors.`**SIGINT_TRACKER** `= []`

The `logger.py` module

Summary

Classes

PyGeometryCustomAdapter	Keeps the reference to the Geometry service instance name dynamic.
PyGeometryPercentStyle	Provides a common messaging style for the `PyGeometryFormatter` class.
PyGeometryFormatter	Provides a `Formatter` class for overwriting default format styles.
InstanceFilter	Ensures that the `instance_name` record always exists.
Logger	Provides the logger used for each PyAnsys Geometry session.

Functions

addfile_handler	Add a file handler to the input.
add_stdout_handler	Add a standout handler to the logger.

Attributes

string_to_loglevel

Constants

LOG_LEVEL
FILE_NAME
DEBUG
INFO
WARN
ERROR
CRITICAL
STDOUT_MSG_FORMAT
FILE_MSG_FORMAT
DEFAULT_STDOUT_HEADER
DEFAULT_FILE_HEADER
NEW_SESSION_HEADER
LOG

PyGeometryCustomAdapter

class PyGeometryCustomAdapter(*logger*, *extra=None*)

Bases: `logging.LoggerAdapter`

Keeps the reference to the Geometry service instance name dynamic.

Overview

Methods

process	Process the logging message and keyword arguments passed in to
log_to_file	Add a file handler to the logger.
log_to_stdout	Add a standard output handler to the logger.
setLevel	Change the log level of the object and the attached handlers.

Attributes

level
file_handler
stdout_handler

Import detail

```
from ansys.geometry.core.logger import PyGeometryCustomAdapter
```

Attribute detail

PyGeometryCustomAdapter.**level**

PyGeometryCustomAdapter.**file_handler**

PyGeometryCustomAdapter.**stdout_handler**

Method detail

PyGeometryCustomAdapter.**process**(*msg*, *kwargs*)

Process the logging message and keyword arguments passed in to a logging call to insert contextual information. You can either manipulate the message itself, the keyword args or both. Return the message and kwargs modified (or not) to suit your needs.

Normally, you'll only need to override this one method in a LoggerAdapter subclass for your specific needs.

PyGeometryCustomAdapter.**log_to_file**(*filename: str = FILE_NAME*, *level: int = LOG_LEVEL*)

Add a file handler to the logger.

Parameters

filename
[str, default: "pyansys-geometry.log"] Name of the file to write log messages to.

level
[int, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

PyGeometryCustomAdapter.**log_to_stdout**(*level=LOG_LEVEL*)

Add a standard output handler to the logger.

Parameters

level
[int, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

PyGeometryCustomAdapter.**setLevel**(*level='DEBUG'*)

Change the log level of the object and the attached handlers.

Parameters

level
[int, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

PyGeometryPercentStyle

class PyGeometryPercentStyle(*fmt*, *, *defaults=None*)

Bases: `logging.PercentStyle`

Provides a common messaging style for the `PyGeometryFormatter` class.

Import detail

```
from ansys.geometry.core.logger import PyGeometryPercentStyle
```

PyGeometryFormatter

class PyGeometryFormatter(*fmt=STDOUT_MSG_FORMAT*, *datefmt=None*, *style='%'*, *validate=True*, *defaults=None*)

Bases: `logging.Formatter`

Provides a `Formatter` class for overwriting default format styles.

Import detail

```
from ansys.geometry.core.logger import PyGeometryFormatter
```

InstanceFilter

class InstanceFilter(*name=''*)

Bases: `logging.Filter`

Ensures that the `instance_name` record always exists.

Overview

Methods

filter	Ensure that the `instance_name` attribute is always present.

Import detail

```
from ansys.geometry.core.logger import InstanceFilter
```

Method detail

`InstanceFilter.`**`filter`**(*record*)

Ensure that the `instance_name` attribute is always present.

Logger

`class Logger`(*level=logging.DEBUG*, *to_file=False*, *to_stdout=True*, *filename=FILE_NAME*)

Provides the logger used for each PyAnsys Geometry session.

Overview

Methods

log_to_file	Add a file handler to the logger.
log_to_stdout	Add the standard output handler to the logger.
setLevel	Change the log level of the object and the attached handlers.
add_child_logger	Add a child logger to the main logger.
add_instance_logger	Add a logger for a Geometry service instance.
add_handling_uncaught_expections	Redirect the output of an exception to a logger.

Attributes

file_handler
std_out_handler

Special methods

__getitem__	Overload the access method by item for the `Logger` class.

Import detail

```
from ansys.geometry.core.logger import Logger
```

Attribute detail

`Logger.`**`file_handler`**

`Logger.`**`std_out_handler`**

Method detail

`Logger.`**`log_to_file`**(*filename=FILE_NAME*, *level=LOG_LEVEL*)

Add a file handler to the logger.

Parameters

filename
[`str`, default: "pyansys-geometry.log"] Name of the file to write log messages to.

level
[`int`, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

Examples

Write to the `"pyansys-geometry.log"` file in the current working directory:

```
>>> from ansys.geometry.core import LOG
>>> import os
>>> file_path = os.path.join(os.getcwd(), 'pyansys-geometry.log')
>>> LOG.log_to_file(file_path)
```

`Logger.`**`log_to_stdout`**(*level=LOG_LEVEL*)

Add the standard output handler to the logger.

Parameters

level
[`int`, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

`Logger.`**`setLevel`**(*level='DEBUG'*)

Change the log level of the object and the attached handlers.

`Logger.`**`add_child_logger`**(*sufix: str*, *level: beartype.typing.Optional[str] = None*)

Add a child logger to the main logger.

This logger is more general than an instance logger, which is designed to track the state of Geometry service instances.

If the logging level is in the arguments, a new logger with a reference to the `_global` logger handlers is created instead of a child logger.

Parameters

sufix
[str] Name of the child logger.

level
[str, default: None] Level of logging.

Returns

logging.Logger
Logger class.

Logger.**add_instance_logger**(*name: str*, *client_instance:* ansys.geometry.core.connection.client.GrpcClient, *level: beartype.typing.Optional[int] = None*) → *PyGeometryCustomAdapter*

Add a logger for a Geometry service instance.

The Geometry service instance logger is a logger with an adapter that adds contextual information such as the Geometry service instance name. This logger is returned, and you can use it to log events as a normal logger. It is stored in the `_instances` field.

Parameters

name
[str] Name for the new instance logger.

client_instance
[*GrpcClient*] Geometry service GrpcClient object, which should contain the `get_name` method.

level
[int, default: None] Level of logging.

Returns

PyGeometryCustomAdapter
Logger adapter customized to add Geometry service information to the logs. You can use this class to log events in the same way you would with the `Logger` class.

Logger.**__getitem__**(*key*)

Overload the access method by item for the `Logger` class.

Logger.**add_handling_uncaught_expections**(*logger*)

Redirect the output of an exception to a logger.

Parameters

logger
[str] Name of the logger.

Description

Provides a general framework for logging in PyAnsys Geometry.

This module is built on the Logging facility for Python. It is not intended to replace the standard Python logging library but rather provide a way to interact between its `logging` class and PyAnsys Geometry.

The loggers used in this module include the name of the instance, which is intended to be unique. This name is printed in all active outputs and is used to track the different Geometry service instances.

Logger usage

Global logger

There is a global logger named `PyAnsys_Geometry_global` that is created when `ansys.geometry.core.__init__` is called. If you want to use this global logger, you must call it at the top of your module:

```
from ansys.geometry.core import LOG
```

You can rename this logger to avoid conflicts with other loggers (if any):

```
from ansys.geometry.core import LOG as logger
```

The default logging level of `LOG` is `ERROR`. You can change this level and output lower-level messages with this code:

```
LOG.logger.setLevel("DEBUG")
LOG.file_handler.setLevel("DEBUG")  # If present.
LOG.stdout_handler.setLevel("DEBUG")  # If present.
```

Alternatively, you can ensure that all the handlers are set to the input log level with this code:

```
LOG.setLevel("DEBUG")
```

This logger does not log to a file by default. If you want, you can add a file handler with this code:

```
import os

file_path = os.path.join(os.getcwd(), "pyansys-geometry.log")
LOG.log_to_file(file_path)
```

This also sets the logger to be redirected to this file. If you want to change the characteristics of this global logger from the beginning of the execution, you must edit the `__init__` file in the directory `ansys.geometry.core`.

To log using this logger, call the desired method as a normal logger with:

```
>>> import logging
>>> from ansys.geometry.core.logging import Logger
>>> LOG = Logger(level=logging.DEBUG, to_file=False, to_stdout=True)
>>> LOG.debug("This is LOG debug message.")

DEBUG -  -  <ipython-input-24-80df150fe31f> - <module> - This is LOG debug message.
```

Instance logger

Every time an instance of the *`Modeler`* class is created, a logger is created and stored in `LOG._instances`. This field is a dictionary where the key is the name of the created logger.

These instance loggers inherit the `PyAnsys_Geometry_global` output handlers and logging level unless otherwise specified. The way this logger works is very similar to the global logger. If you want to add a file handler, you can use the *`log_to_file()`* method. If you want to change the log level, you can use the `setLevel()` method.

Here is an example of how you can use this logger:

```
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler()
>>> modeler._log.info("This is a useful message")

INFO - GRPC_127.0.0.1:50056 -  <...> - <module> - This is a useful message
```

Other loggers

You can create your own loggers using a Python `logging` library as you would do in any other script. There would be no conflicts between these loggers.

Module detail

`logger.`**`addfile_handler`**`(`*logger*, *filename=FILE_NAME*, *level=LOG_LEVEL*, *write_headers=False*`)`

Add a file handler to the input.

Parameters

logger
[`logging.Logger`] Logger to add the file handler to.

filename
[`str`, default: "pyansys-geometry.log"] Name of the output file.

level
[`int`, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

write_headers
[bool, default: `False`] Whether to write the headers to the file.

Returns

Logger
Logger or `logging.Logger` object.

`logger.`**`add_stdout_handler`**`(`*logger*, *level=LOG_LEVEL*, *write_headers=False*`)`

Add a standout handler to the logger.

Parameters

logger
[`logging.Logger`] Logger to add the file handler to.

level
[`int`, default: 10] Level of logging. The default is `10`, in which case the `logging.DEBUG` level is used.

write_headers
[bool, default: `False`] Whether to write headers to the file.

Returns

Logger
Logger or `logging.Logger` object.

`logger.`**`LOG_LEVEL`**

`logger.FILE_NAME = 'pyansys-geometry.log'`

`logger.DEBUG`

`logger.INFO`

`logger.WARN`

`logger.ERROR`

`logger.CRITICAL`

`logger.STDOUT_MSG_FORMAT = '%(levelname)s - %(instance_name)s - %(module)s - %(funcName)s - %(message)s'`

`logger.FILE_MSG_FORMAT`

`logger.DEFAULT_STDOUT_HEADER = Multiline-String`

```
"""
LEVEL - INSTANCE NAME - MODULE - FUNCTION - MESSAGE
"""
```

`logger.DEFAULT_FILE_HEADER`

`logger.NEW_SESSION_HEADER`

`logger.LOG`

`logger.string_to_loglevel`

The `modeler.py` module

Summary

Classes

`Modeler`	Provides for interacting with an open session of the Geometry service.

Modeler

class Modeler(*host: str = DEFAULT_HOST*, *port: beartype.typing.Union[str, int] = DEFAULT_PORT*, *channel: beartype.typing.Optional[grpc.Channel] = None*, *remote_instance: beartype.typing.Optional[ansys.platform.instancemanagement.Instance] = None*, *local_instance: beartype.typing.Optional[*ansys.geometry.core.connection.local_instance.LocalDockerInstance*] = None*, *product_instance: beartype.typing.Optional[*ansys.geometry.core.connection.product_instance.ProductInstance*] = None*, *timeout: beartype.typing.Optional[ansys.geometry.core.typing.Real] = 120*, *logging_level: beartype.typing.Optional[int] = logging.INFO*, *logging_file: beartype.typing.Optional[beartype.typing.Union[pathlib.Path, str]] = None*, *backend_type: beartype.typing.Optional[*ansys.geometry.core.connection.backend.BackendType*] = None*)

Provides for interacting with an open session of the Geometry service.

Overview

Methods

create_design	Initialize a new design with the connected client.
read_existing_design	Read the existing design on the service with the connected client.
close	`Modeler` method for easily accessing the client's close method.
open_file	Open a file.
run_discovery_script_file	Run a Discovery script file.

Properties

client	`Modeler` instance client.
repair_tools	Access to repair tools.

Special methods

__repr__	Represent the modeler as a string.

Import detail

```
from ansys.geometry.core.modeler import Modeler
```

Property detail

property `Modeler.`**client:** ***GrpcClient***

`Modeler` instance client.

property `Modeler.`**repair_tools:** ***RepairTools***

Access to repair tools.

Method detail

`Modeler.`**create_design**(*name: str*) → *ansys.geometry.core.designer.design.Design*

Initialize a new design with the connected client.

Parameters

name
[`str`] Name for the new design.

Returns

Design
Design object created on the server.

`Modeler.read_existing_design()` → *ansys.geometry.core.designer.design.Design*

Read the existing design on the service with the connected client.

Returns

Design
Design object already existing on the server.

`Modeler.close()` → None

`Modeler` method for easily accessing the client's close method.

`Modeler.open_file`(*file_path: str*, *upload_to_server: bool = True*, *import_options:* ansys.geometry.core.misc.options.ImportOptions = *ImportOptions()*) → *ansys.geometry.core.designer.design.Design*

Open a file.

This method imports a design into the service. On Windows, `.scdocx` and HOOPS Exchange formats are supported. On Linux, only the `.scdocx` format is supported.

If the file is a shattered assembly with external references, the whole containing folder will need to be uploaded. Ensure proper folder structure in order to prevent the uploading of unnecessary files.

Parameters

file_path
[`str`] Path of the file to open. The extension of the file must be included.

upload_to_server
[bool] True if the service is running on a remote machine. If service is running on the local machine, set to False, as there is no reason to upload the file.

import_options
[*`ImportOptions`*] Import options that toggle certain features when opening a file.

Returns

Design
Newly imported design.

`Modeler.__repr__()` → str

Represent the modeler as a string.

`Modeler.run_discovery_script_file`(*file_path: str*, *script_args: beartype.typing.Dict[str, str]*, *import_design=False*) → beartype.typing.Tuple[beartype.typing.Dict[str, str], beartype.typing.Optional[*ansys.geometry.core.designer.design.Design*]]

Run a Discovery script file.

The implied API version of the script should match the API version of the running Geometry Service. DMS API versions 23.2.1 and later are supported. DMS is a Windows-based modeling service that has been containerized to ease distribution, execution, and remotability operations.

Parameters

file_path
[`str`] Path of the file. The extension of the file must be included.

script_args
[`dict`[`str`, `str`]] Arguments to pass to the script.

import_design
[bool, default: `False`] Whether to refresh the current design from the service. When the

script is expected to modify the existing design, set this to `True` to retrieve up-to-date design data. When this is set to `False` (default) and the script modifies the current design, the design may be out-of-sync.

Returns

`dict[str, str]`
Values returned from the script.

***`Design`*, `optional`**
Up-to-date current design. This is only returned if `import_design=True`.

Raises

`GeometryRuntimeError`
If the Discovery script fails to run. Otherwise, assume that the script ran successfully.

Description

Provides for interacting with the Geometry service.

The `typing.py` module

Summary

Attributes

`Real`	Type used to refer to both integers and floats as possible values.
`RealSequence`	Type used to refer to `Real` types as a `Sequence` type.

Description

Provides typing of values for PyAnsys Geometry.

Module detail

`typing.`**`Real`**
Type used to refer to both integers and floats as possible values.

`typing.`**`RealSequence`**
Type used to refer to `Real` types as a `Sequence` type.

Notes

`numpy.ndarrays` are also accepted because they are the overlaying data structure behind most PyAnsys Geometry objects.

3.1.2 Description

PyAnsys Geometry is a Python wrapper for the Ansys Geometry service.

3.1.3 Module detail

`core.USE_TRAME = False`

Global constant for checking whether to use trame for visualization.

`core.__version__`

PyAnsys Geometry version.

CHAPTER

FOUR

EXAMPLES

These examples demonstrate the behavior and usage of PyAnsys Geometry.

4.1 PyAnsys Geometry 101 examples

These examples demonstrate basic operations you can perform with PyAnsys Geometry.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.1.1 PyAnsys Geometry 101: Math

The `math` module is the foundation of PyAnsys Geometry. This module is built on top of NumPy, one of the most renowned mathematical Python libraries.

This example shows some of the main PyAnsys Geometry math objects and demonstrates why they are important prior to doing more exciting things in PyAnsys Geometry.

Perform required imports

Perform the required imports.

[1]:
```python
import numpy as np

from ansys.geometry.core.math import Plane, Point2D, Point3D, Vector2D, Vector3D,␣
↪UnitVector3D
```

Create points and vectors

Everything starts with Point and Vector objects, which can each be defined in a 2D or 3D form. These objects inherit from NumPy's ndarray, providing them with enhanced functionalities. When creating these objects, you must remember to pass in the arguments as a list (that is, with brackets [ ]).

Create 2D and 3D point and vectors.

```
Point3D([x, y, z])
Point2D([x, y])

Vector3D([x, y, z])
Vector2D([x, y])
```

You can perform standard mathematical operations on points and vectors.

Perform some standard operations on vectors.

[2]:
```
vec_1 = Vector3D([1,0,0]) # x-vector
vec_2 = Vector3D([0,1,0]) # y-vector

print("Sum of vectors [1, 0, 0] + [0, 1, 0]:")
print(vec_1 + vec_2) # sum

print("\nDot product of vectors [1, 0, 0] * [0, 1, 0]:")
print(vec_1 * vec_2) # dot

print("\nCross product of vectors [1, 0, 0] % [0, 1, 0]:")
print(vec_1 % vec_2) # cross
```

```
Sum of vectors [1, 0, 0] + [0, 1, 0]:
[1 1 0]

Dot product of vectors [1, 0, 0] * [0, 1, 0]:
0

Cross product of vectors [1, 0, 0] % [0, 1, 0]:
[0 0 1]
```

Create a vector from two points.

[3]:
```
p1 = Point3D([12.4, 532.3, 89])
p2 = Point3D([-5.7, -67.4, 46.6])

vec_3 = Vector3D.from_points(p1, p2)
vec_3
```

[3]:
```
Vector3D([ -18.1, -599.7,  -42.4])
```

Normalize a vector to create a unit vector, which is also known as a *direction*.

[4]:
```
print("Magnitude of vec_3:")
print(vec_3.magnitude)

print("\nNormalized vec_3:")
print(vec_3.normalize())
```

(continues on next page)

(continued from previous page)

```
print("\nNew magnitude:")
print(vec_3.normalize().magnitude)
```

```
Magnitude of vec_3:
601.4694173438911

Normalized vec_3:
[-0.03009297 -0.99705818 -0.07049402]

New magnitude:
1.0
```

Use the `UnitVector` class to automatically normalize the input for the unit vector.

```
[5]: uv = UnitVector3D([1,1,1])
     uv
```

```
[5]: UnitVector3D([0.57735027, 0.57735027, 0.57735027])
```

Perform a few more mathematical operations on vectors.

```
[6]: v1 = Vector3D([1, 0, 0])
     v2 = Vector3D([0, 1, 0])

     print("Vectors are perpendicular:")
     print(v1.is_perpendicular_to(v2))

     print("\nVectors are parallel:")
     print(v1.is_parallel_to(v2))

     print("\nVectors are opposite:")
     print(v1.is_opposite(v2))

     print("\nAngle between vectors:")
     print(v1.get_angle_between(v2))
     print(f"{np.pi / 2} == pi/2")
```

```
Vectors are perpendicular:
True

Vectors are parallel:
False

Vectors are opposite:
False

Angle between vectors:
1.5707963267948966 radian
1.5707963267948966 == pi/2
```

Create planes

Once you begin creating sketches and bodies, `Plane` objects become very important. A plane is defined by these items:

- An origin, which consists of a 3D point
- Two directions (`direction_x` and `direction_y`), which are both `UnitVector3D`objects

If no direction vectors are provided, the plane defaults to the XY plane.

Create two planes.

[7]:
```
plane = Plane(Point3D([0,0,0])) # XY plane

print("(1, 2, 0) is in XY plane:")
print(plane.is_point_contained(Point3D([1, 2, 0]))) # True

print("\n(0, 0, 5) is in XY plane:")
print(plane.is_point_contained(Point3D([0, 0, 5]))) # False
```

```
(1, 2, 0) is in XY plane:
True

(0, 0, 5) is in XY plane:
False
```

Perform parametric evaluations

PyAnsys Geometry implements parametric evaluations for some curves and surfaces.

Evaluate a sphere.

[8]:
```
from ansys.geometry.core.primitives.sphere import Sphere, SphereEvaluation
from ansys.geometry.core.math import Point3D
from ansys.geometry.core.misc import Distance

sphere = Sphere(Point3D([0,0,0]), Distance(1)) # radius = 1

eval = sphere.project_point(Point3D([1,1,1]))

print("U Parameter:")
print(eval.parameter.u)

print("\nV Parameter:")
print(eval.parameter.v)
```

```
U Parameter:
0.7853981633974483

V Parameter:
0.6154797086703873
```

[9]:
```
print("Point on the sphere:")
eval.position
```

```
Point on the sphere:
```

[9]:
```
Point3D([0.57735027, 0.57735027, 0.57735027])
```

[10]:
```python
print("Normal to the surface of the sphere at the evaluation position:")
eval.normal
```

```
Normal to the surface of the sphere at the evaluation position:
```

[10]:
```
UnitVector3D([0.57735027, 0.57735027, 0.57735027])
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.1.2 PyAnsys Geometry 101: Units

To handle units inside the source code, PyAnsys Geometry uses Pint, a third-party open source software that other PyAnsys libraries also use.

The following code examples show how to operate with units inside the PyAnsys Geometry codebase and create objects with different units.

Import units handler

The following line of code imports the units handler: `pint.util.UnitRegistry`. For more information on the `UnitRegistry` class in the `pint` API, see Most important classes in the Pint documentation.

[1]:
```python
from ansys.geometry.core.misc import UNITS
```

Create and work with `Quantity` objects

With the `UnitRegistry` object called `UNITS`, you can create `Quantity` objects. A `Quantity` object is simply a container class with two core elements:

- A number
- A unit

`Quantity` objects have convenience methods, including those for transforming to different units and comparing magnitudes, values, and units. For more information on the `Quantity` class in the `pint` API, see Most important classes in the Pint documentation. You can also step through this Pint tutorial.

[2]:
```python
from pint import Quantity

a = Quantity(10, UNITS.mm)

print(f"Object a is a pint.Quantity: {a}")

print("Request its magnitude in different ways (accessor methods):")
print(f"Magnitude: {a.m}.")
print(f"Also magnitude: {a.magnitude}.")

print("Request its units in different ways (accessor methods):")
print(f"Units: {a.u}.")
print(f"Also units: {a.units}.")

# Quantities can be compared between different units
# You can also build Quantity objects as follows:
a2 = 10 * UNITS.mm
print(f"Compare quantities built differently: {a == a2}")

# Quantities can be compared between different units
a2_diff_units = 1 * UNITS.cm
print(f"Compare quantities with different units: {a == a2_diff_units}")
```

```
Object a is a pint.Quantity: 10 millimeter
Request its magnitude in different ways (accessor methods):
Magnitude: 10.
Also magnitude: 10.
Request its units in different ways (accessor methods):
Units: millimeter.
Also units: millimeter.
Compare quantities built differently: True
Compare quantities with different units: True
```

PyAnsys Geometry objects work by returning `Quantity` objects whenever the property requested has a physical meaning.

Return `Quantity` objects for `Point3D` objects.

[3]:
```python
from ansys.geometry.core.math import Point3D

point_a = Point3D([1,2,4])
print("========================= Point3D([1,2,4]) ========================")
print(f"Point3D is a numpy.ndarray in SI units: {point_a}.")
print(f"However, request each of the coordinates individually...\n")
print(f"X Coordinate: {point_a.x}")
print(f"Y Coordinate: {point_a.y}")
print(f"Z Coordinate: {point_a.z}\n")

# Now, store the information with different units...
point_a_km = Point3D([1,2,4], unit=UNITS.km)
print("================= Point3D([1,2,4], unit=UNITS.km) =================")
print(f"Point3D is a numpy.ndarray in SI units: {point_a_km}.")
print(f"However, request each of the coordinates individually...\n")
print(f"X Coordinate: {point_a_km.x}")
```

(continues on next page)

(continued from previous page)

```
print(f"Y Coordinate: {point_a_km.y}")
print(f"Z Coordinate: {point_a_km.z}\n")

# These points, although they are in different units, can be added together.
res = point_a + point_a_km

print("================== res = point_a + point_a_km ====================")
print(f"numpy.ndarray: {res}")
print(f"X Coordinate: {res.x}")
print(f"Y Coordinate: {res.y}")
print(f"Z Coordinate: {res.z}")
```

```
========================= Point3D([1,2,4]) ========================
Point3D is a numpy.ndarray in SI units: [1. 2. 4.].
However, request each of the coordinates individually...

X Coordinate: 1 meter
Y Coordinate: 2 meter
Z Coordinate: 4 meter

================= Point3D([1,2,4], unit=UNITS.km) =================
Point3D is a numpy.ndarray in SI units: [1000. 2000. 4000.].
However, request each of the coordinates individually...

X Coordinate: 1 kilometer
Y Coordinate: 2 kilometer
Z Coordinate: 4 kilometer

=================== res = point_a + point_a_km ====================
numpy.ndarray: [1001. 2002. 4004.]
X Coordinate: 1001.0 meter
Y Coordinate: 2002.0 meter
Z Coordinate: 4004.0 meter
```

Use default units

PyAnsys Geometry implements the concept of *default units*.

[4]:
```
from ansys.geometry.core.misc import DEFAULT_UNITS

print("=== Default unit length ===")
print(DEFAULT_UNITS.LENGTH)

print("=== Default unit angle ===")
print(DEFAULT_UNITS.ANGLE)
```

```
=== Default unit length ===
meter
=== Default unit angle ===
radian
```

It is important to differentiate between *client-side* default units and *server-side* default units. You are able to control both of them.

Print the default server unit length.

```
[5]: print("=== Default server unit length ===")
     print(DEFAULT_UNITS.SERVER_LENGTH)
```

```
=== Default server unit length ===
meter
```

Use default units.

```
[6]: from ansys.geometry.core.math import Point2D
     from ansys.geometry.core.misc import DEFAULT_UNITS

     DEFAULT_UNITS.LENGTH = UNITS.mm

     point_2d_default_units = Point2D([3, 4])
     print("This is a Point2D with default units")
     print(f"X Coordinate: {point_2d_default_units.x}")
     print(f"Y Coordinate: {point_2d_default_units.y}")
     print(f"numpy.ndarray value: {point_2d_default_units}")

     # Revert back to original default units
     DEFAULT_UNITS.LENGTH = UNITS.m
```

```
This is a Point2D with default units
X Coordinate: 3 millimeter
Y Coordinate: 4 millimeter
numpy.ndarray value: [0.003 0.004]
```

PyAnsys Geometry has certain auxiliary classes implemented that provide proper unit checking when assigning values. Although they are basically intended for internal use of the library, you can define them for use.

```
[7]: from ansys.geometry.core.misc import Angle, Distance
```

Start with `Distance`. The main difference between a `Quantity` object (that is, `from pint import Quantity`) and a `Distance` is that there is an active check on the units passed (in case they are not the default ones). Here are some examples.

```
[8]: radius = Distance(4)
     print(f"The radius is {radius.value}.")

     # Reassign the value of the distance
     radius.value = 7 * UNITS.cm
     print(f"After reassignment, the radius is {radius.value}.")


     # Change the units if desired
     radius.unit = UNITS.cm
     print(f"After changing its units, the radius is {radius.value}.")
```

```
The radius is 4 meter.
After reassignment, the radius is 0.07 meter.
After changing its units, the radius is 7.000000000000001 centimeter.
```

The next two code examples show how unreasonable operations raise errors.

[9]:
```python
try:
    radius.value = 3 * UNITS.degrees
except TypeError as err:
    print(f"Error raised: {err}")
```

```
Error raised: The pint.Unit provided as an input should be a [length] quantity.
```

[10]:
```python
try:
    radius.unit = UNITS.fahrenheit
except TypeError as err:
    print(f"Error raised: {err}")
```

```
Error raised: The pint.Unit provided as an input should be a [length] quantity.
```

The same behavior applies to the `Angle` object. Here are some examples.

[11]:
```python
import numpy as np

rotation_angle = Angle(np.pi / 2)
print(f"The rotation angle is {rotation_angle.value}.")

# Try reassigning the value of the distance
rotation_angle.value = 7 * UNITS.degrees
print(f"After reassignment, the rotation angle is {rotation_angle.value}.")

# You could also change its units if desired
rotation_angle.unit = UNITS.degrees
print(f"After changing its units, the rotation angle is {rotation_angle.value}.")
```

```
The rotation angle is 1.5707963267948966 radian.
After reassignment, the rotation angle is 0.12217304763960307 radian.
After changing its units, the rotation angle is 7.0 degree.
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.1.3 PyAnsys Geometry 101: Sketching

With PyAnsys Geometry, you can build powerful dynamic sketches without communicating with the Geometry service. This example shows how to build some simple sketches.

Perform required imports

Perform the required imports.

```
[1]: from pint import Quantity

from ansys.geometry.core.math import Plane, Point2D, Point3D, Vector3D
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.sketch import Sketch
```

Add a box to sketch

The `Sketch` object is the starting point. Once it is created, you can dynamically add various curves to the sketch. Here are some of the curves that are available:

- `arc`
- `box`
- `circle`
- `ellipse`
- `gear`
- `polygon`
- `segment`
- `slot`
- `trapezoid`
- `triangle`

Add a box to the sketch.

```
[2]: sketch = Sketch()

sketch.segment(Point2D([0,0]), Point2D([0,1]))
sketch.segment(Point2D([0,1]), Point2D([1,1]))
sketch.segment(Point2D([1,1]), Point2D([1,0]))
sketch.segment(Point2D([1,0]), Point2D([0,0]))

sketch.plot()
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
```

```
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

A *functional-style sketching API* is also implemented. It allows you to append curves to the sketch with the idea of *never picking up your pen*.

Use the functional-style sketching API to add a box.

```
[3]: sketch = Sketch()

(
    sketch.segment(Point2D([0,0]), Point2D([0,1]))
        .segment_to_point(Point2D([1,1]))
        .segment_to_point(Point2D([1,0]))
        .segment_to_point(Point2D([0,0]))
)

sketch.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

A `Sketch` object uses the XY plane by default. You can define your own custom plane using three parameters: `origin`, `direction_x`, and `direction_y`.

Add a box on a custom plane.

```
[4]: plane = Plane(origin=Point3D([0,0,0]), direction_x=Vector3D([1,2,-1]), direction_
↪y=Vector3D([1,0,1]))

sketch = Sketch(plane)

sketch.box(Point2D([0,0]), 1, 1)

sketch.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Combine concepts to create powerful sketches

Combine these simple concepts to create powerful sketches.

```
[5]: # Complex Fluent API Sketch - PCB

sketch = Sketch()

(
```

(continues on next page)

(continued from previous page)

```
    sketch.segment(Point2D([0, 0], unit=UNITS.mm), Point2D([40, 1], unit=UNITS.mm),
↪"LowerEdge")
        .arc_to_point(Point2D([41.5, 2.5], unit=UNITS.mm), Point2D([40, 2.5], unit=UNITS.
↪mm), tag="SupportedCorner")
        .segment_to_point(Point2D([41.5, 5], unit=UNITS.mm))
        .arc_to_point(Point2D([43, 6.5], unit=UNITS.mm), Point2D([43, 5], unit=UNITS.mm),␣
↪True)
        .segment_to_point(Point2D([55, 6.5], unit=UNITS.mm))
        .arc_to_point(Point2D([56.5, 8], unit=UNITS.mm), Point2D([55, 8], unit=UNITS.mm))
        .segment_to_point(Point2D([56.5, 35], unit=UNITS.mm))
        .arc_to_point(Point2D([55, 36.5], unit=UNITS.mm), Point2D([55, 35], unit=UNITS.mm))
        .segment_to_point(Point2D([0, 36.5], unit=UNITS.mm))
        .segment_to_point(Point2D([0, 0], unit=UNITS.mm))
        .circle(Point2D([4, 4], UNITS.mm), Quantity(1.5, UNITS.mm), "Anchor1")
        .circle(Point2D([51, 34.5], UNITS.mm), Quantity(1.5, UNITS.mm), "Anchor2")
)

sketch.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.1.4 PyAnsys Geometry 101: Modeling

Once you understand PyAnsys Geometry's mathematical constructs, units, and sketching capabilities, you can dive into its modeling capabilities.

PyAnsys Geometry is a Python client that connects to a modeling service. Here are the modeling services that are available for connection:

- **DMS**: Windows-based modeling service that has been containerized to ease distribution, execution, and remotability operations.
- **Geometry service**: Linux-based approach of DMS that is currently under development.
- **Ansys Discovery and SpaceClaim**: PyAnsys Geometry is capable of connecting to a running session of Ansys Discovery or SpaceClaim. Although this is not the main use case for PyAnsys Geometry, a connection to one of these Ansys products is possible. Because these products have graphical user interfaces, performance is not as high with this option as with the previous options. However, connecting to a running instance of Discovery or SpaceClaim might be useful for some users.

Launch a modeling service

While the PyAnsys Geometry operations in earlier examples did not require communication with a modeling service, this example requires that a modeling service is available. All subsequent examples also require that a modeling service is available.

Launch a modeling service session.

[1]:
```
from ansys.geometry.core import launch_modeler

# Start a modeler session
modeler = launch_modeler()
print(modeler)
```

```
Ansys Geometry Modeler (0x20df223f730)

Ansys Geometry Modeler Client (0x20df223f8e0)
  Target:     localhost:700
  Connection: Healthy
```

You can also launch your own services and connect to them. For information on connecting to an existing service, see the Modeler API documentation.

Here is how the class architecture is implemented:

- `Modeler`: Handler object for the active service session. This object allows you to connect to an existing service by passing in a host and a port. It also allows you to create `Design` objects, which is where the modeling takes place. For more information, see the Modeler API documentation.
- `Design`: Root object of your assembly (tree). While a `Design` object is also a `Component` object, it has enhanced capabilities, including creating named selections, adding materials, and handling beam profiles. For more information, see the Design API documentation.
- `Component`: One of the main objects for modeling purposes. `Component` objects allow you to create bodies, subcomponents, beams, design points, planar surfaces, and more. For more information, see the Component API documentation.

The following code examples show how you use these objects. More capabilities of these objects are shown in the specific example sections for sketching and modeling.

Create and plot a sketch

Create a `Sketch` object and plot it.

[2]:
```
from ansys.geometry.core.sketch import Sketch
from ansys.geometry.core.math import Point2D
from ansys.geometry.core.misc import UNITS, Distance

outer_hole_radius = Distance(0.5, UNITS.m)

sketch = Sketch()
(
    sketch.segment(start=Point2D([-4, 5], unit=UNITS.m), end=Point2D([4, 5], unit=UNITS.
↪m))
    .segment_to_point(end=Point2D([4, -5], unit=UNITS.m))
    .segment_to_point(end=Point2D([-4, -5], unit=UNITS.m))
```

(continues on next page)

(continued from previous page)

```
    .segment_to_point(end=Point2D([-4, 5], unit=UNITS.m))
    .box(
        center=Point2D([0, 0], unit=UNITS.m),
        width=Distance(3, UNITS.m),
        height=Distance(3, UNITS.m),
    )
    .circle(center=Point2D([3, 4], unit=UNITS.m), radius=outer_hole_radius)
    .circle(center=Point2D([-3, -4], unit=UNITS.m), radius=outer_hole_radius)
    .circle(center=Point2D([-3, 4], unit=UNITS.m), radius=outer_hole_radius)
    .circle(center=Point2D([3, -4], unit=UNITS.m), radius=outer_hole_radius)
)

# Plot the sketch
sketch.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Perform some modeling operations

Now that the sketch is ready to be extruded, perform some modeling operations, including creating the design, creating the body directly on the design, and plotting the body.

[3]:
```
# Start by creating the Design
design = modeler.create_design("ModelingDemo")

# Create a body directly on the design by extruding the sketch
body = design.extrude_sketch(
    name="Design_Body", sketch=sketch, distance=Distance(80, unit=UNITS.cm)
)

# Plot the body
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Perform some operations on the body

Perform some operations on the body.

```
[4]: # Request its faces, edges, volume...
faces = body.faces
edges = body.edges
volume = body.volume

print(f"This is body {body.name} with ID (server-side): {body.id}.")
print(f"This body has {len(faces)} faces and {len(edges)} edges.")
print(f"The body volume is {volume}.")
```

```
This is body Design_Body with ID (server-side): 0:22.
This body has 14 faces and 32 edges.
The body volume is 54.28672587712814 meter ** 3.
```

Other operations that can be performed include adding a midsurface offset and thickness (only for planar bodies), imprinting curves, assigning materials, copying, and translating.

Copy the body on a new subcomponent and translate it.

```
[5]: from ansys.geometry.core.math import UNITVECTOR3D_X

# Create a component
comp = design.add_component("Component")

# Copy the body that belongs to this new component
body_copy = body.copy(parent=comp, name="Design_Component_Body")

# Displace this new body by a certain distance (10m) in a certain direction (X-axis)
body_copy.translate(direction=UNITVECTOR3D_X, distance=Distance(10, unit=UNITS.m))

# Plot the result of the entire design
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Create and assign materials to the bodies that were created.

```
[6]: from pint import Quantity

from ansys.geometry.core.materials import Material, MaterialProperty,␣
↪MaterialPropertyType

# Define some general properties for the material.
density = Quantity(125, 10 * UNITS.kg / (UNITS.m * UNITS.m * UNITS.m))
poisson_ratio = Quantity(0.33, UNITS.dimensionless)
tensile_strength = Quantity(45)  # WARNING: If no units are defined,
#it is assumed that the magnitude is in the units expected by the server.

# Once your material properties are defined, you can easily create a material.
material = Material(
    "steel",
```

(continues on next page)

(continued from previous page)

```
    density,
    [MaterialProperty(MaterialPropertyType.POISSON_RATIO, "PoissonRatio", poisson_
↪ratio)],
)

# If you forgot to add a property, or you want to overwrite its value, you can still
# add properties to your created material.
material.add_property(
    type=MaterialPropertyType.TENSILE_STRENGTH, name="TensileProp", quantity=tensile_
↪strength
)

# Once your material is properly defined, send it to the server.
# This material can then be reused by different objects
design.add_material(material)

# Assign your material to your existing bodies.
body.assign_material(material)
body_copy.assign_material(material)
```

Currently materials do not have any impact on the visualization when plotting is requested, although this could be a future feature. If the final assembly is open in Discovery or SpaceClaim, you can observe the changes.

Create a named selection

PyAnsys Geometry supports the creation of a named selection via the `Design` object.

Create a named selection with some of the faces of the previous body and the body itself.

[7]:
```
# Create a named selection
faces = body.faces
ns = design.create_named_selection("MyNamedSelection", bodies=[body], faces=[faces[0],␣
↪faces[-1]])
print(f"This is a named selection called {ns.name} with ID (server-side): {ns.id}.")
```

```
This is a named selection called MyNamedSelection with ID (server-side): 0:429.
```

Perform deletions

Deletion operations for bodies, named selections, and components are possible, always from the scope expected. For example, if you attempted to delete the original body from a component that has no ownership over it (such as your `comp` object), the deletion would fail. If you attempted to perform this deletion from the `design` object, the deletion would succeed.

The next two code examples show how deletion works.

[8]:
```
# If you try to delete this body from an "unauthorized" component, the deletion is not␣
↪allowed.
comp.delete_body(body)
print(f"Is the body alive? {body.is_alive}")

```

(continues on next page)

(continued from previous page)

```
# If you request a plot of the entire design, you can still see it.
design.plot()
```

```
Is the body alive? True
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

[9]:
```
# Because the body belongs to the ``design`` object and not the ``comp`` object,
# deleting it from ``design`` object works.
design.delete_body(body)
print(f"Is the body alive? {body.is_alive}")

# If you request a plot of the entire design, it is no longer visible.
design.plot()
```

```
Is the body alive? False
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Export files

Once modeling operations are finalized, you can export files in different formats. For the formats supported by DMS, see the DesignFileFormat class in the `Design` module documentation.

Export files in SCDOCX and FMD formats.

[10]:
```
import os
from pathlib import Path

from ansys.geometry.core.designer import DesignFileFormat

# Path to downloads directory
file_dir = Path(os.getcwd(), "downloads")
file_dir.mkdir(parents=True, exist_ok=True)

# Download the model in different formats
design.download(file_location=Path(file_dir, "ModelingDemo.scdocx"),␣
↪format=DesignFileFormat.SCDOCX)
design.download(file_location=Path(file_dir, "ModelingDemo.fmd"),␣
↪format=DesignFileFormat.FMD)
```

Close session

When you finish interacting with your modeling service, you should close the active server session. This frees resources wherever the service is running.

Close the server session.

[11]:
```
modeler.close()
```

Note: If the server session already existed (that is, it was not launched by the current client session), you cannot use this method to close the server session. You must manually close the server serssion instead. This is a safeguard for user-spawned services.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.1.5 PyAnsys Geometry 101: Plotter

This example provides an overview of PyAnsys Geometry's plotting capabilities, focusing on its plotter features. After reviewing the fundamental concepts of sketching and modeling in PyAnsys Geometry, it shows how to leverage these key plotting capabilities:

- **Multi-object plotting**: You can conveniently plot a list of elements, including objects created in both PyAnsys Geometry and PyVista libraries.
- **Interactive object selection**: You can interactively select PyAnsys Geometry objects within the scene. This enables efficient manipulation of these objects in subsequent scripting.

Perform required imports

Perform the required imports.

[1]:
```
from pint import Quantity
import pyvista as pv

from ansys.geometry.core import Modeler
from ansys.geometry.core.connection.defaults import GEOMETRY_SERVICE_DOCKER_IMAGE
from ansys.geometry.core.connection.local_instance import LocalDockerInstance
from ansys.geometry.core.math import Point2D
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.plotting import PlotterHelper
from ansys.geometry.core.sketch import Sketch
```

Load modeling service

Load the modeling service. While the following code uses a Docker image to interact with the modeling service, you can use any suitable method mentioned in the preceding examples.

[2]:
```
list_images = []
list_containers = []
available_images = LocalDockerInstance.docker_client().images.list(
    name=GEOMETRY_SERVICE_DOCKER_IMAGE
)
is_image_available_cont = None
for image in available_images:
    for geom_image, geom_cont in zip(list_images, list_containers):
        if geom_image in image.tags:
            is_image_available = True
            is_image_available_cont = geom_cont
            break

local_instance = LocalDockerInstance(
    connect_to_existing_service=True,
    restart_if_existing_service=True,
    image=is_image_available_cont,
)

modeler = Modeler(local_instance=local_instance)
```

Instantiate design and initialize object list

Instantiate a new design to work on and initialize a list of objects for plotting.

[3]:
```
# init modeler
design = modeler.create_design("Multiplot")

plot_list = []
```

You are now ready to create some objects and use the plotter capabilities.

Create a PyAnsys Geometry body cylinder

Use PyAnsys Geometry to create a body cylinder.

[4]:
```
cylinder = Sketch()
cylinder.circle(Point2D([10, 10], UNITS.m), 1.0)
cylinder_body = design.extrude_sketch("JustACyl", cylinder, Quantity(10, UNITS.m))
plot_list.append(cylinder_body)
```

Create a PyAnsys Geometry arc sketch

Use PyAnsys Geometry to create an arc sketch.

```
[5]: sketch = Sketch()
sketch.arc(
    Point2D([20, 20], UNITS.m),
    Point2D([20, -20], UNITS.m),
    Point2D([10, 0], UNITS.m),
    tag="Arc",
)
plot_list.append(sketch)
```

Create a PyVista cylinder

Use PyVista to create a cylinder.

```
[6]: cyl = pv.Cylinder(radius=5, height=20, center=(-20, 10, 10))
plot_list.append(cyl)
```

Create a PyVista multiblock

Use PyVista to create a multiblock with a sphere and a cube.

```
[7]: blocks = pv.MultiBlock(
    [pv.Sphere(center=(20, 10, -10), radius=10), pv.Cube(x_length=10, y_length=10, z_
↪length=10)]
)
plot_list.append(blocks)
```

Create a PyAnsys Geometry body box

Use PyAnsys Geometry to create a body box that is a cube.

```
[8]: box2 = Sketch()
box2.box(Point2D([-10, 20], UNITS.m), Quantity(10, UNITS.m), Quantity(10, UNITS.m))
box_body2 = design.extrude_sketch("JustABox", box2, Quantity(10, UNITS.m))
plot_list.append(box_body2)
```

Plot objects

When plotting the created objects, you have several options.

You can simply plot one of the created objects.

```
[9]: plotter = PlotterHelper()
plotter.plot(box_body2)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
```

```
Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

[9]:
```
[]
```

You can plot the whole list of objects.

[10]:
```python
plotter = PlotterHelper()
plotter.plot(plot_list)
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

[10]:
```
[]
```

The Python visualizer is used by default. However, you can also use trame for visualization.

```python
plotter = PlotterHelper(use_trame=True)
plotter.plot(plot_list)
```

Select objects interactively

PyAnsys Geometry's plotter supports interactive object selection within the scene. This enables you to pick objects for subsequent script manipulation.

[11]:
```python
plotter = PlotterHelper(allow_picking=True)

# Plotter returns picked bodies
picked_list = plotter.plot(plot_list)
print(picked_list)
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

```
[]
```

Close session

When you finish interacting with your modeling service, you should close the active server session. This frees resources wherever the service is running.

Close the server session.

[12]:
```
modeler.close()
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.2 Sketching examples

These examples demonstrate math operations on geometric objects and sketching capabilities, combined with server-based operations.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.2.1 Sketching: Basic usage

This example shows how to use basic PyAnsys Geometry sketching capabilities.

Perform required imports

Perform the required imports.

[1]:
```
from ansys.geometry.core.misc.units import UNITS as u
from ansys.geometry.core.sketch import Sketch
from ansys.geometry.core.plotting import Plotter
```

Create a sketch

Sketches are fundamental objects for drawing basic shapes like lines, segments, circles, ellipses, arcs, and polygons.

You create a `Sketch` instance by defining a drawing plane. To define a plane, you declare a point and two fundamental orthogonal directions.

[2]:
```
from ansys.geometry.core.math import Plane, Point2D, Point3D
```

Define a plane for creating a sketch.

[3]:
```
# Define the origin point of the plane
origin = Point3D([1, 1, 1])

# Create a plane located in previous point with desired fundamental directions
plane = Plane(
    origin, direction_x=[1, 0, 0], direction_y=[0, -1, 1]
)

# Instantiate a new sketch object from previous plane
sketch = Sketch(plane)
```

Draw shapes

To draw different shapes in the sketch, you use `draw` methods.

Draw a circle

You draw a circle in a sketch by specifying the center and radius.

[4]:
```
sketch.circle(Point2D([2, 1]), radius=30 * u.cm, tag="Circle")
sketch.select("Circle")
sketch.plot_selection()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Draw an ellipse

You draw an ellipse in a sketch by specifying the center, major radius, and minor radius.

[5]:
```
sketch.ellipse(
    Point2D([1, 1]), major_radius=2*u.m, minor_radius=1*u.m, tag="Ellipse"
)
sketch.select("Ellipse")
sketch.plot_selection()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Draw a polygon

You draw a regular polygon by specifying the center, radius, and desired number of sides.

```
[6]: sketch.polygon(
         Point2D([1, 1]), inner_radius=3*u.m, sides=5, tag="Polygon"
     )
     sketch.select("Polygon")
     sketch.plot_selection()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Draw an arc

You draw an arc of circumference by specifying the center, starting point, and ending point.

```
[7]: start_point, end_point = Point2D([2, 1], unit=u.m), Point2D([0, 1], unit=u.meter)
     sketch.arc(start_point, end_point, Point2D([1,1]), tag="Arc")
     sketch.select("Arc")
     sketch.plot_selection()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Draw a slot

You draw a slot by specifying the center, width, and height.

```
[8]: sketch.slot(Point2D([2, 0]), 4, 3, tag="Slot")
     sketch.select("Slot")
     sketch.plot_selection()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Draw a box

You draw a box by specifying the center, width, and height.

```
[9]: sketch.box(Point2D([2, 0]), 4, 5, tag="Box")
     sketch.select("Box")
     sketch.plot_selection()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Draw a segment

You draw a segment by specifying the starting point and ending point.

```
[10]: start_point, end_point = Point2D([2, 1], unit=u.m), Point2D([0, 1], unit=u.meter)
sketch.segment(start_point, end_point, "Segment")
sketch.select("Segment")
sketch.plot_selection()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Plot the sketch

The Plotter class provides capabilities for plotting different PyAnsys Geometry objects. PyAnsys Geometry uses PyVista as the visualization backend.

You use the plot_sketch method to plot a sketch. This method accepts a Sketch instance and some extra arguments to further customize the visualization of the sketch. These arguments include showing the plane of the sketch and its frame.

```
[11]: # Plot the sketch in the whole scene
pl = Plotter()
pl.plot_sketch(sketch, show_plane=True, show_frame=True)
pl.scene.show(jupyter_backend="panel")
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.2.2 Sketching: Dynamic sketch plane

The sketch is a lightweight, two-dimensional modeler driven primarily by client-side execution.

At any point, the current state of a sketch can be used for operations such as extruding a body, projecting a profile, or imprinting curves.

The sketch is designed as an effective *functional-style* API with all operations receiving 2D configurations.

For easy reuse of sketches across different regions of your design, you can move a sketch around the global coordinate system by modifying the plane defining the current sketch location.

This example creates a multi-layer PCB from many extrusions of the same sketch, creating unique design bodies for each layer.

Perform required imports

Perform the required imports.

```
[1]: from pint import Quantity

from ansys.geometry.core import Modeler
from ansys.geometry.core.math import UNITVECTOR3D_Z, Point2D
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.sketch import Sketch
```

Define sketch profile

You can create, modify, and plot `Sketch` instances independent of supporting Geometry service instances.

To define the sketch profile for the PCB, you create a sketch outline of individual `Segment` and `Arc` objects with two circular through-hole attachment points added within the profile boundary to maintain a single, closed sketch face.

Create a single `Sketch` instance to use for multiple design operations.

```
[2]: sketch = Sketch()

(
    sketch.segment(Point2D([0, 0], unit=UNITS.mm), Point2D([40, 1], unit=UNITS.mm),
↪"LowerEdge")
        .arc_to_point(Point2D([41.5, 2.5], unit=UNITS.mm), Point2D([40, 2.5], unit=UNITS.
↪mm), tag="SupportedCorner")
        .segment_to_point(Point2D([41.5, 5], unit=UNITS.mm))
        .arc_to_point(Point2D([43, 6.5], unit=UNITS.mm), Point2D([43, 5], unit=UNITS.mm),␣
↪True)
        .segment_to_point(Point2D([55, 6.5], unit=UNITS.mm))
        .arc_to_point(Point2D([56.5, 8], unit=UNITS.mm), Point2D([55, 8], unit=UNITS.mm))
        .segment_to_point(Point2D([56.5, 35], unit=UNITS.mm))
        .arc_to_point(Point2D([55, 36.5], unit=UNITS.mm), Point2D([55, 35], unit=UNITS.mm))
        .segment_to_point(Point2D([0, 36.5], unit=UNITS.mm))
        .segment_to_point(Point2D([0, 0], unit=UNITS.mm))
        .circle(Point2D([4, 4], UNITS.mm), Quantity(1.5, UNITS.mm), "Anchor1")
        .circle(Point2D([51, 34.5], UNITS.mm), Quantity(1.5, UNITS.mm), "Anchor2")
)

sketch.plot()
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
```

```
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
```

```
Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Extrude multiple bodies

Establish a server connection and use the single sketch profile to extrude the board profile at multiple Z-offsets. Create a named selection from the resulting list of layer bodies.

Note that translating the sketch plane prior to extrusion is more effective (10 server calls) than creating a design body on the supporting server and then translating the body on the server (20 server calls).

[3]:
```python
modeler = Modeler()
design = modeler.create_design("ExtrudedBoardProfile")

layers = []
layer_thickness = Quantity(0.20, UNITS.mm)
for layer_index in range(10):
    layers.append(design.extrude_sketch(f"BoardLayer_{layer_index}", sketch, layer_
↪thickness))
    sketch.translate_sketch_plane_by_distance(UNITVECTOR3D_Z, layer_thickness)

board_named_selection = design.create_named_selection("FullBoard", bodies=layers)
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.2.3 Sketching: Parametric sketching for gears

This example shows how to use gear sketching shapes from PyAnsys Geometry.

Perform required imports and pre-sketching operations

Perform required imports and instantiate the `Modeler` instance and the basic elements that define a sketch.

[1]:
```python
from pint import Quantity

from ansys.geometry.core import Modeler
from ansys.geometry.core.math import Plane, Point2D, Point3D
from ansys.geometry.core.misc import UNITS, Distance
from ansys.geometry.core.sketch import Sketch
from ansys.geometry.core.plotting import Plotter

# Start a modeler session
modeler = Modeler()

# Define the origin point of the plane
origin = Point3D([1, 1, 1])

# Create a plane containing the previous point with desired fundamental directions
plane = Plane(
    origin, direction_x=[1, 0, 0], direction_y=[0, -1, 1]
)
```

Sketch a dummy gear

`DummyGear` sketches are simple gears that have straight teeth. While they do not ensure actual physical functionality, they might be useful for some simple playground tests.

Instantiate a new `Sketch` object and then define and plot a dummy gear.

[2]:
```python
# Instantiate a new sketch object from previous plane
sketch = Sketch(plane)

# Define dummy gear
#
origin = Point2D([0, 1], unit=UNITS.meter)
outer_radius = Distance(4, unit=UNITS.meter)
inner_radius = Distance(3.8, unit=UNITS.meter)
n_teeth = 30
sketch.dummy_gear(origin, outer_radius, inner_radius, n_teeth)

# Plot dummy gear
sketch.plot()
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
```

```
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

After creating the sketch, extrudes it.

[3]:
```
# Create a design
design = modeler.create_design("AdvancedFeatures_DummyGear")

# Extrude your sketch
dummy_gear = design.extrude_sketch("DummyGear", sketch, Distance(1000, UNITS.mm))

# Plot the design
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Sketch a spur gear

`SpurGear` sketches are parametric CAD spur gears based on four parameters:

- `origin`: Center point location for the desired spur gear. The value must be a `Point2D` object.
- `module`: Ratio between the pitch circle diameter in millimeters and the number of teeth. This is a common parameter for spur gears. The value should be an integer or a float.
- `pressure_angle`: Pressure angle expected for the teeth of the spur gear. This is also a common parameter for spur gears. The value must be a `pint.Quantity` object.
- `n_teeth`: Number of teeth. The value must be an integer.

Instantiate a new `Sketch` object and then define and plot a spur gear.

[4]:
```
# Instantiate a new sketch object from previous plane
sketch = Sketch(plane)

# Define spur gear
#
origin = Point2D([0, 1], unit=UNITS.meter)
module = 40
pressure_angle = Quantity(20, UNITS.deg)
n_teeth = 22

# Sketch spur gear
sketch.spur_gear(origin, module, pressure_angle, n_teeth)

# Plot spur gear
sketch.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

After creating the sketch, extrude it.

```
[5]: # Create a design
design = modeler.create_design("AdvancedFeatures_SpurGear")

# Extrude sketch
dummy_gear = design.extrude_sketch("SpurGear", sketch, Distance(200, UNITS.mm))

# Plot design
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.3 Modeling examples

These examples demonstrate service-based modeling operations.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.3.1 Modeling: Single body with material assignment

In PyAnsys Geometry, a *body* represents solids or surfaces organized within the `Design` assembly. The current state of sketch, which is a client-side execution, can be used for the operations of the geometric design assembly.

The Geometry service provides data structures to create individual materials and their properties. These data structures are exposed through PyAnsys Geometry.

This example shows how to create a single body from a sketch by requesting its extrusion. It then shows how to assign a material to this body.

Perform required imports

Perform the required imports.

```
[1]: from pint import Quantity

from ansys.geometry.core import Modeler
from ansys.geometry.core.materials import Material, MaterialProperty,␣
↪MaterialPropertyType
from ansys.geometry.core.math import UNITVECTOR3D_Z, Frame, Plane, Point2D, Point3D,␣
↪UnitVector3D
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.sketch import Sketch
```

Create sketch

Create a Sketch instance and insert a circle with a radius of 10 millimeters in the default plane.

```
[2]: sketch = Sketch()
sketch.circle(Point2D([10, 10], UNITS.mm), Quantity(10, UNITS.mm))
```

```
[2]: <ansys.geometry.core.sketch.sketch.Sketch at 0x2374b311bb0>
```

Initiate design on server

Establish a server connection and initiate a design on the server.

```
[3]: modeler = Modeler()
design_name = "ExtrudeProfile"
design = modeler.create_design(design_name)
```

Add materials to design

Add materials and their properties to the design. Material properties can be added when creating the Material object or after its creation. This code adds material properties after creating the Material object.

```
[4]: density = Quantity(125, 10 * UNITS.kg / (UNITS.m * UNITS.m * UNITS.m))
poisson_ratio = Quantity(0.33, UNITS.dimensionless)
tensile_strength = Quantity(45)
material = Material(
    "steel",
    density,
    [MaterialProperty(MaterialPropertyType.POISSON_RATIO, "PoissonRatio", poisson_
↪ratio)],
)
material.add_property(MaterialPropertyType.TENSILE_STRENGTH, "TensileProp", Quantity(45))
design.add_material(material)
```

Extrude sketch to create body

Extrude the sketch to create the body and then assign a material to it.

[5]:
```python
# Extrude the sketch to create the body
body = design.extrude_sketch("SingleBody", sketch, Quantity(10, UNITS.mm))

# Assign a material to the body
body.assign_material(material)

body.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.3.2 Modeling: Rectangular plate with multiple bodies

You can create multiple bodies from a single sketch by extruding the same sketch in different planes.

The sketch is designed as an effective *functional-style* API with all operations receiving 2D configurations. For more information, see the :ref:`Sketch <ref_sketch>` subpackage.

In this example, a box is located in the center of the plate, with the default origin of a sketch plane (origin at `(0, 0, 0)`). Four holes of equal radius are sketched at the corners of the plate. The plate is then extruded, leading to the generation of the requested body. The projection is at the center of the face. The default projection depth is through the entire part.

Perform required imports

Perform the required imports.

```
[1]: import numpy as np
     from pint import Quantity

     from ansys.geometry.core import Modeler
     from ansys.geometry.core.math import Plane, Point3D, Point2D, UnitVector3D
     from ansys.geometry.core.misc import UNITS
     from ansys.geometry.core.sketch import Sketch
```

Define sketch profile

The sketch profile for the proposed design requires four segments that constitute the outer limits of the design, a box on the center, and a circle at its four corners.

You can use a single `sketch` instance for multiple design operations, including extruding a body, projecting a profile, and imprinting curves.

Define the sketch profle for the rectangular plate with multiple bodies.

```
[2]: sketch = Sketch()
     (sketch.segment(Point2D([-4, 5], unit=UNITS.m), Point2D([4, 5], unit=UNITS.m))
         .segment_to_point(Point2D([4, -5], unit=UNITS.m))
         .segment_to_point(Point2D([-4, -5], unit=UNITS.m))
         .segment_to_point(Point2D([-4, 5], unit=UNITS.m))
         .box(Point2D([0,0], unit=UNITS.m), Quantity(3, UNITS.m), Quantity(3, UNITS.m))
         .circle(Point2D([3, 4], unit=UNITS.m), Quantity(0.5, UNITS.m))
         .circle(Point2D([-3, -4], unit=UNITS.m), Quantity(0.5, UNITS.m))
         .circle(Point2D([-3, 4], unit=UNITS.m), Quantity(0.5, UNITS.m))
         .circle(Point2D([3, -4], unit=UNITS.m), Quantity(0.5, UNITS.m))
     )
```

```
[2]: <ansys.geometry.core.sketch.sketch.Sketch at 0x15ca88301c0>
```

Extrude sketch to create design

Establish a server connection and use the single sketch profile to extrude the base component at the Z axis. Create a named selection from the resulting list of bodies. In only three server calls, the design extrudes the four segments with the desired thickness.

```
[3]: modeler = Modeler()
     design = modeler.create_design("ExtrudedPlate")

     body = design.extrude_sketch(f"PlateLayer", sketch, Quantity(2, UNITS.m))

     board_named_selection = design.create_named_selection("Plate", bodies=[body])
     design.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

```
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
```

```
Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Add component with a planar surface

After creating a plate as a base component, you might want to add a component with a planar surface to it.

Create a `sketch` instance and then create a surface in the design with this sketch. For the sketch, it creates an ellipse, keeping the origin of the plane as its center.

[4]:
```
# Add components to the design
planar_component = design.add_component("PlanarComponent")

# Initiate ``Sketch`` to create the planar surface.
planar_sketch = Sketch()
planar_sketch.ellipse(
        Point2D([0, 0], UNITS.m), Quantity(1, UNITS.m), Quantity(0.5, UNITS.m)
    )

planar_body = planar_component.create_surface("PlanarComponentSurface", planar_sketch)

comp_str = repr(planar_component)
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Extrude from face to create body

Extrude a face profile by a given distance to create a solid body. There are no modifications against the body containing the source face.

[5]:
```
longer_body = design.extrude_face(
        "LongerEllipseFace", planar_body.faces[0], Quantity(5, UNITS.m)
    )
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Translate body within plane

Use the :func:`translate()<ansys.geometry.core.designer.body.Body.translate>` method to move the body in a specified direction by a given distance. You can also move a sketch around the global coordinate system. For more information, see the *Dynamic Sketch Plane* example.

[6]:
```
longer_body.translate(UnitVector3D([1, 0, 0]), Quantity(4, UNITS.m))
design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.3.3 Modeling: Tessellation of two bodies

This example shows how to create two stacked bodies and return the tessellation as two merged bodies.

Perform required imports

Perform the required imports.

[1]:
```
from pint import Quantity

from ansys.geometry.core import Modeler
from ansys.geometry.core.math import Point2D, Point3D, Plane
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.plotting import Plotter
from ansys.geometry.core.sketch import Sketch
```

Create design

Create the basic sketches to be tessellated and extrude the sketch in the required plane. For more information on creating a component and extruding a sketch in the design, see the *Rectangular plate with multiple bodies* example.

Here is a typical situation in which two bodies, with different sketch planes, merge each body into a single dataset. This effectively combines all the faces of each individual body into a single dataset without separating faces.

[2]:
```
modeler = Modeler()

sketch_1 = Sketch()
box = sketch_1.box(
    Point2D([10, 10], unit=UNITS.m), width=Quantity(10, UNITS.m), height=Quantity(5,␣
↪UNITS.m)
)
circle = sketch_1.circle(
    Point2D([0, 0], unit=UNITS.m), radius=Quantity(25, UNITS.m)
)

design = modeler.create_design("TessellationDesign")
comp = design.add_component("TessellationComponent")
body = comp.extrude_sketch("Body", sketch=sketch_1, distance=10 * UNITS.m)

# Create the second body in a plane with a different origin
sketch_2 = Sketch(Plane([0, 0, 10]))
box = sketch_2.box(Point2D(
    [10, 10], unit=UNITS.m), width=Quantity(10, UNITS.m), height=Quantity(5, UNITS.m)
)
circle = sketch_2.circle(
    Point2D([0, 10], unit=UNITS.m), radius=Quantity(25, UNITS.m)
)

body = comp.extrude_sketch("Body", sketch=sketch_2, distance=10 * UNITS.m)
```

Tessellate component as two merged bodies

Tessellate the component and merge each body into a single dataset. This effectively combines all the faces of each individual body into a single dataset without separating faces.

[3]:
```
dataset = comp.tessellate(merge_bodies=True)
dataset
```

[3]:
```
MultiBlock (0x27097a53280)
  N Blocks    1
  X Bounds    -25.000, 25.000
  Y Bounds    -24.999, 34.999
  Z Bounds    0.000, 20.000
```

If you want to tessellate the body and return the geometry as triangles, single body tessellation is possible. If you want to merge the individual faces of the tessellation, enable the `merge` option so that the body is rendered into a single mesh. This preserves the number of triangles and only merges the topology.

Code without merging the body

[4]:
```
dataset = body.tessellate()
dataset
```

[4]:
```
MultiBlock (0x270a3586520)
  N Blocks    7
  X Bounds    -25.000, 25.000
  Y Bounds    -14.999, 34.999
  Z Bounds    10.000, 20.000
```

Code with merging the body

```
[5]: mesh = body.tessellate(merge=True)
mesh
```

```
[5]: PolyData (0x270a35af220)
  N Cells:    1640
  N Points:   1650
  N Strips:   0
  X Bounds:   -2.500e+01, 2.500e+01
  Y Bounds:   -1.500e+01, 3.500e+01
  Z Bounds:   1.000e+01, 2.000e+01
  N Arrays:   0
```

Plot design

Plot the design.

```
[6]: design.plot()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.3.4 Modeling: Design organization

The Design instance creates a design project within the remote Geometry service to complete all CAD modeling against.

You can organize all solid and surface bodies in each design within a customizable component hierarchy. A component is simply an organization mechanism.

The top-level design node and each child component node can have one or more bodies assigned and one or more components assigned.

The API requires each component of the design hierarchy to be given a user-defined name.

There are several design operations that result in a body being created within a design. Executing each of these methods against a specific component instance explicitly specifies the node of the design tree to place the new body under.

Perform required imports

Perform the required imports.

```
[1]: from ansys.geometry.core import Modeler
     from ansys.geometry.core.math import UNITVECTOR3D_X, Point2D
     from ansys.geometry.core.misc import UNITS, Distance
     from ansys.geometry.core.sketch import Sketch
```

Organize design

Extrude two sketches to create bodies. Assign the cylinder to the top-level design component. Assign the slot to the component nested one level beneath the top-level design component.

```
[2]: modeler = Modeler()

     design = modeler.create_design("DesignHierarchyExample")

     circle_sketch = Sketch()
     circle_sketch.circle(Point2D([10, 10], UNITS.mm), Distance(10, UNITS.mm))

     cylinder_body = design.extrude_sketch("10mmCylinder", circle_sketch, Distance(10, UNITS.
     ↪mm))

     slot_sketch = Sketch()
     slot_sketch.slot(Point2D([40, 10], UNITS.mm), Distance(20, UNITS.mm), Distance(10, UNITS.
     ↪mm))

     nested_component = design.add_component("NestedComponent")
     slot_body = nested_component.extrude_sketch("SlotExtrusion", slot_sketch, Distance(20,␣
     ↪UNITS.mm))

     design.plot()
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript
```

```
Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Create nested component

Create a component that is nested under the previously created component and then create another cylinder from the previously used sketch.

[3]:
```python
double_nested_component = nested_component.add_component("DoubleNestedComponent")

circle_surface_body = double_nested_component.create_surface("CircularSurfaceBody",␣
↪circle_sketch)
circle_surface_body.translate(UNITVECTOR3D_X, Distance(-35, UNITS.mm))

design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Use surfaces from body to create additional bodies

You can use surfaces from any body across the entire design as references for creating additional bodies.

Extrude a cylinder from the surface body assigned to the child component.

[4]:
```python
cylinder_from_face = nested_component.extrude_face("CylinderFromFace", circle_surface_
↪body.faces[0], Distance(30, UNITS.mm))
cylinder_from_face.translate(UNITVECTOR3D_X, Distance(-25, UNITS.mm))

design.plot()
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

Download this example

Download this example as a Jupyter Notebook or as a Python script.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

4.3.5 Modeling: Boolean operations

This example shows how to use Boolean operations for geometry manipulation.

Perform required imports

Perform the required imports.

```
[1]: from typing import List

from ansys.geometry.core import launch_local_modeler
from ansys.geometry.core.designer import Body
from ansys.geometry.core.math import Point2D
from ansys.geometry.core.misc import UNITS
from ansys.geometry.core.plotting import PlotterHelper
from ansys.geometry.core.sketch import Sketch
```

Launch local modeler

Launch the local modeler. If you are not familiar with how to launch the local modeler, see the "Launch a modeling service" section in the *PyAnsys Geometry 101: Modeling* example.

```
[2]: modeler = launch_local_modeler()
```

Define bodies

This section defines the bodies to use the Boolean operations on. First you create sketches of a box and a circle, and then you extrude these sketches to create 3D objects.

Create sketches

Create sketches of a box and a circle that serve as the basis for your bodies.

```
[3]: # Create a sketch of a box
sketch_box = Sketch().box(Point2D([0, 0], unit=UNITS.m), width=30 * UNITS.m, height=40 *␣
↪UNITS.m)

# Create a sketch of a circle (overlapping the box slightly)
sketch_circle = Sketch().circle(Point2D([20, 0], unit=UNITS.m), radius=10 * UNITS.m)
```

Extrude sketches

After the sketches are created, extrude them to create 3D objects.

[4]:
```
# Create a design
design = modeler.create_design("example_design")

# Extrude both sketches to get a prism and a cylinder
prism = design.extrude_sketch("Prism", sketch_box, 50 * UNITS.m)
cylin = design.extrude_sketch("Cylinder", sketch_circle, 50 * UNITS.m)
```

You must extrude the sketches each time that you perform an example operation. This is because performing a Boolean operation modifies the underlying design permanently. Thus, you no longer have two bodies. As shown in the Boolean operations themselves, whenever you pass in a body, it is consumed, and so it no longer exists. The remaining body (with the performed Boolean operation) is the one that performed the call to the method.

Select bodies

You can optionally select bodies in the plotter as described in the "Select objects interactively" section in the *PyAnsys Geometry 101: Plotter* example. As shown in this example, the plotter preserves the picking order, meaning that the output list is sorted according to the picking order.

```
bodies: List[Body] = PlotterHelper(allow_picking=True).plot(design.bodies)
```

Otherwise, you can select bodies from the design directly.

[5]:
```
bodies = [design.bodies[0], design.bodies[1]]
```

Perform Boolean operations

This section performs Boolean operations on the defined bodies using the PyAnsys Geometry library. It explores intersection, union, and subtraction operations.

Perform an intersection operation

To perform an intersection operation on the bodies, first set up the bodies.

[6]:
```
# Create a design
design = modeler.create_design("intersection_design")

# Extrude both sketches to get a prism and a cylinder
prism = design.extrude_sketch("Prism", sketch_box, 50 * UNITS.m)
cylin = design.extrude_sketch("Cylinder", sketch_circle, 50 * UNITS.m)
```

Perform the intersection and plot the results.

[7]:
```
prism.intersect(cylin)
_ = PlotterHelper().plot(design.bodies)
```

```
Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json
```

Data type cannot be displayed: application/vnd.holoviews_load.v0+json, application/javascript

Data type cannot be displayed: text/html, application/vnd.holoviews_exec.v0+json

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

The final remaining body is the `prism` body because the `cylin` body has been consumed.

[8]:
```
print(design.bodies)
```

```
[
ansys.geometry.core.designer.Body 0x1e3f0467130
  Name                   : Prism
  Exists                 : True
  Parent component       : intersection_design
  MasterBody             : 0:22
  Surface body           : False
]
```

Perform a union operation

To carry out a union operation on the bodies, first set up the bodies.

[9]:
```
# Create a design
design = modeler.create_design("union_design")

# Extrude both sketches to get a prism and a cylinder
prism = design.extrude_sketch("Prism", sketch_box, 50 * UNITS.m)
cylin = design.extrude_sketch("Cylinder", sketch_circle, 50 * UNITS.m)
```

Perform the union and plot the results.

[10]:
```
prism.unite(cylin)
_ = PlotterHelper().plot(design.bodies)
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

The final remaining body is the `prism` body because the `cylin` body has been consumed.

[11]:
```
print(design.bodies)
```

```
[
ansys.geometry.core.designer.Body 0x1e3f0467b20
  Name                   : Prism
  Exists                 : True
  Parent component       : union_design
  MasterBody             : 0:22
```

(continues on next page)

(continued from previous page)

```
  Surface body          : False
]
```

Perform a subtraction operation

To perform a subtraction operation on the bodies, first set up the bodies.

[12]:
```
# Create a design
design = modeler.create_design("subtraction_design")

# Extrude both sketches to get a prism and a cylinder
prism = design.extrude_sketch("Prism", sketch_box, 50 * UNITS.m)
cylin = design.extrude_sketch("Cylinder", sketch_circle, 50 * UNITS.m)
```

Perform the subtraction and plot the results.

[13]:
```
prism.subtract(cylin)
_ = PlotterHelper().plot(design.bodies)
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

The final remaining body is the `prism` body because the `cylin` body has been consumed.

[14]:
```
print(design.bodies)
```

```
[
ansys.geometry.core.designer.Body 0x1e3fdd944c0
  Name                  : Prism
  Exists                : True
  Parent component      : subtraction_design
  MasterBody            : 0:22
  Surface body          : False
]
```

If you perform this action inverting the order of the bodies (that is, `cylin.subtract(prism)`, you can see the difference in the resulting shape of the body.

[15]:
```
# Create a design
design = modeler.create_design("subtraction_design_inverted")

# Extrude both sketches to get a prism and a cylinder
prism = design.extrude_sketch("Prism", sketch_box, 50 * UNITS.m)
cylin = design.extrude_sketch("Cylinder", sketch_circle, 50 * UNITS.m)

# Invert subtraction
cylin.subtract(prism)
_ = PlotterHelper().plot(design.bodies)
```

```
VTKRenderWindowSynchronized(vtkWin32OpenGLRenderWindow, orientation_widget=True, sizing_
↪mode='stretch_width')
```

In this case, the final remaining body is the `cylin` body because the `prism` body has been consumed.

[16]:
```python
print(design.bodies)
```

```
[
ansys.geometry.core.designer.Body 0x1e3f043bd60
  Name                   : Cylinder
  Exists                 : True
  Parent component       : subtraction_design_inverted
  MasterBody             : 0:85
  Surface body           : False
]
```

Summary

These Boolean operations provide powerful tools for creating complex geometries and combining or modifying existing shapes in meaningful ways.

Feel free to experiment with different shapes, sizes, and arrangements to further enhance your understanding of Boolean operations in PyAnsys Geometry and their applications.

Download this example

Download this example as a Jupyter Notebook or as a Python script.

CHAPTER

FIVE

CONTRIBUTE

Overall guidance on contributing to a PyAnsys library appears in the Contributing topic in the *PyAnsys Developer's Guide*. Ensure that you are thoroughly familiar with this guide before attempting to contribute to PyAnsys Geometry.

The following contribution information is specific to PyAnsys Geometry.

5.1 Clone the repository

To clone and install the latest PyAnsys Geometry release in development mode, run these commands:

```
git clone https://github.com/ansys/pyansys-geometry
cd pyansys-geometry
python -m pip install --upgrade pip
pip install -e .
```

5.2 Post issues

Use the PyAnsys Geometry Issues page to submit questions, report bugs, and request new features. When possible, you should use these issue templates:

- Bug, problem, error: For filing a bug report
- Documentation error: For requesting modifications to the documentation
- Adding an example: For proposing a new example
- New feature: For requesting enhancements to the code

If your issue does not fit into one of these template categories, you can click the link for opening a blank issue.

To reach the project support team, email pyansys.core@ansys.com.

5.3 View documentation

Documentation for the latest stable release of PyAnsys Geometry is hosted at PyAnsys Geometry Documentation.

In the upper right corner of the documentation's title bar, there is an option for switching from viewing the documentation for the latest stable release to viewing the documentation for the development version or previously released versions.

5.4 Adhere to code style

PyAnsys Geometry follows the PEP8 standard as outlined in PEP 8 in the *PyAnsys Developer's Guide* and implements style checking using pre-commit.

To ensure your code meets minimum code styling standards, run these commands:

```
pip install pre-commit
pre-commit run --all-files
```

You can also install this as a pre-commit hook by running this command:

```
pre-commit install
```

This way, it's not possible for you to push code that fails the style checks:

```
$ pre-commit install
$ git commit -am "added my cool feature"
black....................................................................Passed
blacken-docs.............................................................Passed
isort....................................................................Passed
flake8...................................................................Passed
docformatter.............................................................Passed
codespell................................................................Passed
pydocstyle...............................................................Passed
check for merge conflicts................................................Passed
debug statements (python)................................................Passed
check yaml...............................................................Passed
trim trailing whitespace.................................................Passed
Add License Headers......................................................Passed
Validate GitHub Workflows................................................Passed
```

CHAPTER SIX

ASSETS

In this section, users are able to download a set of assets related to PyAnsys Geometry.

6.1 Documentation

The following links will provide users with downloadable documentation in various formats

- Documentation in HTML format
- Documentation in PDF format

6.2 Wheelhouse

If you lack an internet connection on your installation machine, you should install PyAnsys Geometry by downloading the wheelhouse archive.

Each wheelhouse archive contains all the Python wheels necessary to install PyAnsys Geometry from scratch on Windows, Linux, and MacOS from Python 3.9 to 3.11. You can install this on an isolated system with a fresh Python installation or on a virtual environment.

For example, on Linux with Python 3.9, unzip the wheelhouse archive and install it with:

```
unzip ansys-geometry-core-v0.4.dev0-wheelhouse-ubuntu-latest3.9.zip wheelhouse
pip install ansys-geometry-core -f wheelhouse --no-index --upgrade --ignore-installed
```

If you are on Windows with Python 3.9, unzip to a wheelhouse directory and install using the preceding command.

Consider installing using a virtual environment.

The following wheelhouse files are available for download:

6.2.1 Linux

- Linux wheelhouse for Python 3.9
- Linux wheelhouse for Python 3.10
- Linux wheelhouse for Python 3.11

6.2.2 Windows

- Windows wheelhouse for Python 3.9
- Windows wheelhouse for Python 3.10
- Windows wheelhouse for Python 3.11

6.2.3 MacOS

- MacOS wheelhouse for Python 3.9
- MacOS wheelhouse for Python 3.10
- MacOS wheelhouse for Python 3.11

6.3 Geometry service Docker container assets

Build the latest Geometry service Docker container using the following assets. Instructions on how to build the containers are found at Docker containers.

Currently, the Geometry service backend is mainly delivered as a **Windows** Docker container. However, these containers require a Windows machine to run them.

A Linux version of the Geometry service is also available but with limited capabilities, meaning that certain operations are not available or fail.

6.3.1 Windows container

Note: Only users with access to https://github.com/ansys/pyansys-geometry-binaries can download these binaries.

- Latest Geometry service binaries for Windows containers
- Latest Geometry service Dockerfile for Windows containers

6.3.2 Linux container

Note: Only users with access to https://github.com/ansys/pyansys-geometry-binaries can download these binaries.

- Latest Geometry service binaries for Linux containers
- Latest Geometry service Dockerfile for Linux containers

PYTHON MODULE INDEX