



© 2024 ANSYS, Inc. or affiliated companies  
Unauthorized use, distribution, or duplication prohibited.

## PyAnsys Geometry

---



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

Mar 04, 2024

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

## **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.

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

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

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

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

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



---

**CHAPTER  
ONE**

---

## **GETTING STARTED**

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

### **1.1 Installation**

You can use `pip` to install PyAnsys Geometry.

```
pip install ansys-geometry-core
```

### **1.2 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.

### **1.3 Compatibility with Ansys releases**

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

## 1.4 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*.

## 1.5 Frequently asked questions

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

### 1.5.1 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.

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.

*Go to Getting started*

#### Windows Docker container

##### Contents

- *Windows Docker container*
  - *Docker for Windows containers*
  - *Build or install the Geometry service image*
    - \* *GitHub Container Registry*
    - \* *Build the Geometry service Windows container*
      - *Prerequisites*
      - *Build from available Ansys installation*
      - *Build the Docker image from available binaries*
  - *Launch the Geometry service*

- \* *Environment variables*
- \* *Geometry service launcher*
- *Connect to the Geometry service*

## 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 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX > 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:

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

There are two build modes:

- **Build from available Ansys installation:** This mode builds the Docker image using the Ansys installation available in the machine where the Docker image is being built.
- **Build from available binaries:** This mode builds the Docker image using the binaries available in the `ansys/pyansys-geometry-binaries` repository. If you do not have access to this repository, you can only use the first mode. Link to the binaries repository: <https://github.com/ansys/pyansys-geometry-binaries/>

## Prerequisites

- Ensure that Docker is installed in your machine. If you do not have Docker available, see *Docker for Windows containers*.

## Build from available Ansys installation

To build your own image based on your own Ansys installation, follow these instructions:

- Download the [Python Docker build script](#).
- Execute the script with the following command (no specific location needed):

```
python build_docker_windows.py
```

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

```
docker images
```

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

## Build the Docker image from available binaries

Prior to building your image, follow these steps:

- 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.

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 windows/Dockerfile .
```

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

```
docker images
```

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

## 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.

```
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_docker_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>
```

**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:

```
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

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

*Go to Docker containers*

*Go to Getting started*

## 1.5.2 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 = launch_modeler_with_discovery()
```

## SpaceClaim

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

## Geometry service

```
from ansys.geometry.core import launch_modeler_with_geometry_service  
  
modeler = 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*

### 1.5.3 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:

```
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:

```
from ansys.discovery.core import launch_modeler_with_pimlight_and_spaceclaim  
  
sc = launch_modeler_with_pimlight_and_spaceclaim("version")
```

(continues on next page)

(continued from previous page)

```
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*

## 1.5.4 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.

**Warning:** Running a SpaceClaim or Discovery normal session does not suffice to be able to use it with PyAnsys Geometry. Both products need the ApiServer extension to be running. In this case, to ease the process, you should launch the products directly from the PyAnsys Geometry library as shown in *Launch a local session*.

### 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=50051)
```

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
Ansyst Geometry Modeler Client (...)
Target:      localhost:50051
Connection:  Healthy
```

*Go to Getting started*

### 1.5.5 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	Space-Claim
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*

### 1.5.6 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

- *Install package in development mode*
  - *Package dependencies*
  - *PyPI*
  - *GitHub*
  - *Install in offline mode*
  - *Verify your installation*

#### 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 3.12. 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)

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*

### 1.5.7 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.

#### How is the Ansys Geometry Service installed?

---

**Note:** This question is answered in <https://github.com/ansys/pyansys-geometry/issues/1022> and <https://github.com/ansys/pyansys-geometry/discussions/883>

---

The Ansys Geometry service is available as a standalone service and it is installed through the Ansys unified installer or the automated installer. Both are available for download from the [Ansys Customer Portal](#).

When using the automated installer, the Ansys Geometry service is installed by default.

However, when using the unified installer, it is necessary to pass in the `-geometrieservice` flag to install it.

Overall, the command to install the Ansys Geometry service with the unified installer is:

```
setup.exe -silent -geometrieservice
```

You can verify that the installation was successful by checking whether the product has been installed on your file directory. If you are using the default installation directory, the product is installed in the following directory:

```
C:\Program Files\ANSYS Inc\vXXX\GeometryService
```

Where `vXXX` is the Ansys version that you have installed.

## What Ansys license is needed 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 SpaceClaim.

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.

## How to build the Docker image for the Ansys Geometry service?

---

**Note:** This question is answered in <https://github.com/ansys/pyansys-geometry/discussions/883>

---

To build your own Docker image for the Ansys Geometry service, users should follow the instructions provided in *Build from available Ansys installation*. The resulting image is a Windows-based Docker image that contains the Ansys Geometry service.

*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:

```
from ansys.geometry.core.math import Plane, Point3D, UnitVector3D

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.

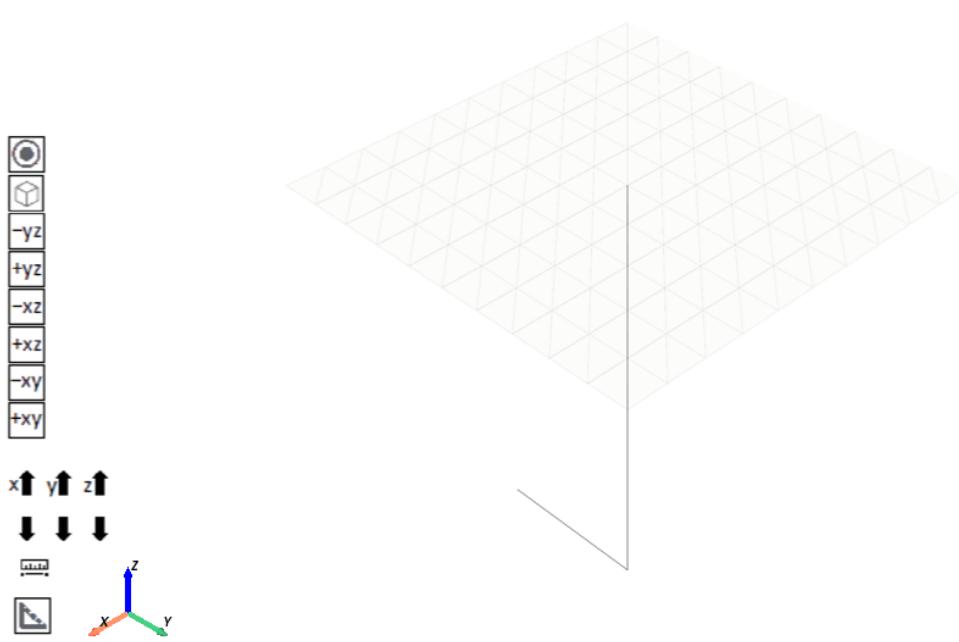
One of the key features of a functional-style API is that it keeps an active context based on the previously created edges to use as a reference starting point for additional objects.

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.

```
from ansys.geometry.core.math import Point2D  
  
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("Segment2")
```

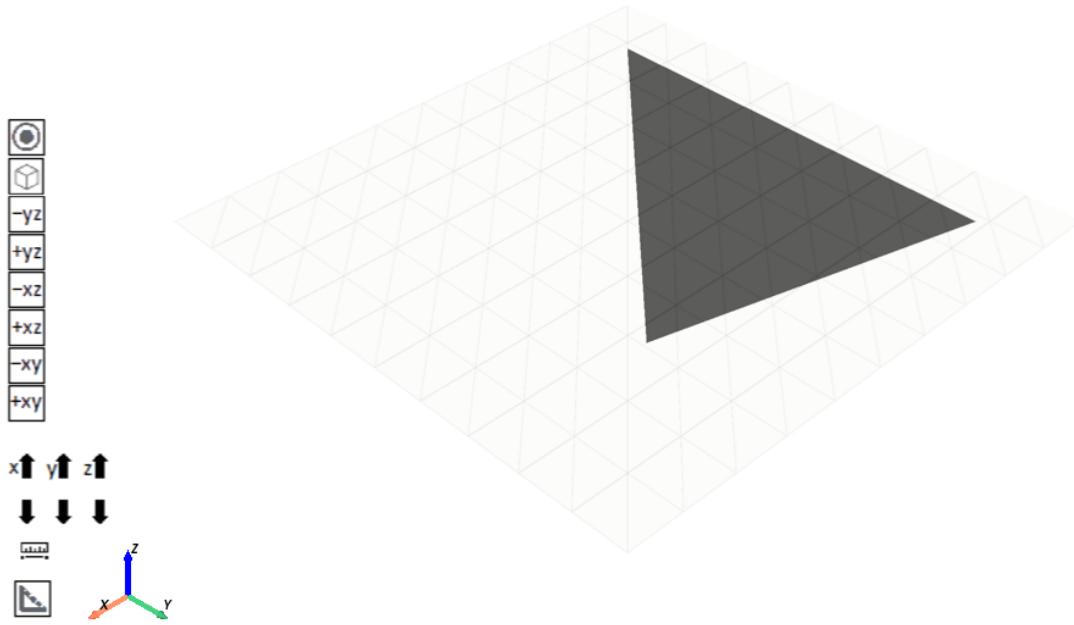


## 2.2.2 Direct API

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()
```



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 Create the model

This code create the *Modeler()* object which owns the whole designs tools and data.

```
from ansys.geometry.core import Modeler

# Create the modeler object itself
modeler = Modeler()
```

### 2.3.2 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.

```
from ansys.geometry.core.sketch import Sketch
from ansys.geometry.core.math import Point2D
from ansys.geometry.core.misc import UNITS
from pint import Quantity

# Create a sketch and draw a circle on the client
sketch = Sketch()
sketch.circle(Point2D([10, 10], UNITS.mm), Quantity(10, UNITS.mm))

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

### 2.3.3 Add materials to model

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

```
from ansys.geometry.core.materials.material import Material
from ansys.geometry.core.materials.property import (
    MaterialProperty,
    MaterialPropertyType,
)

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)
```

### 2.3.4 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))
```

### 2.3.5 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()*

### 2.3.6 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.scdocx"  
design.download(file)
```

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()
```

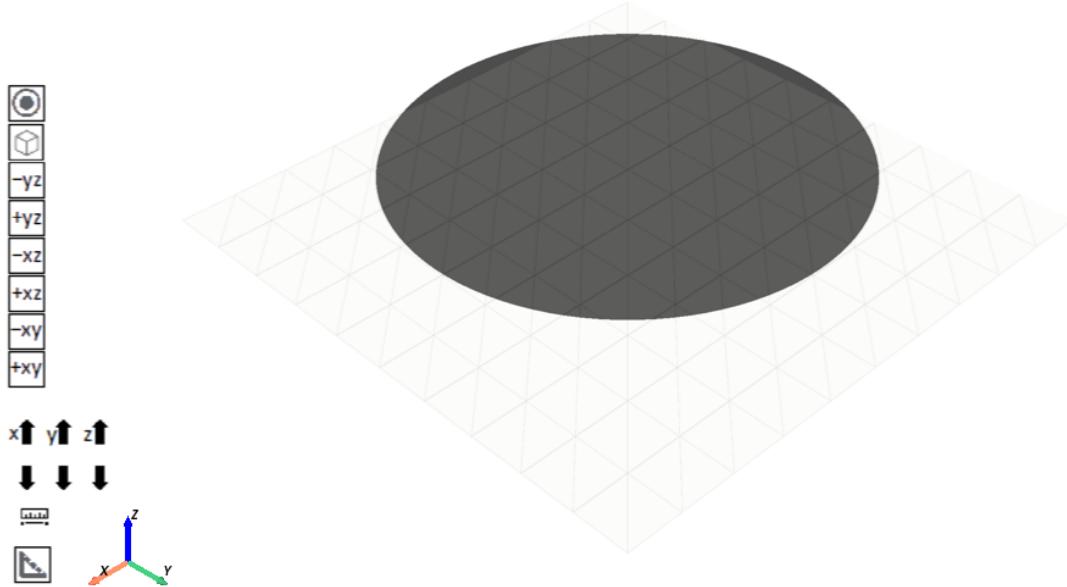
## 2.5.2 Create geometry model

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()
```





## 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

---

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

---

##### Submodules

---

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

---

## Attributes

<code>__version__</code>	PyAnsys Geometry version.
--------------------------	---------------------------

## Constants

<code>USE_TRAME</code>	Global constant for checking whether to use <code>trame</code>
<code>DISABLE_MULTIPLE_DESIGN_CHECK</code>	Global constant for disabling the <code>ensure_design_is_active</code> check.
<code>DOCUMENTATION_BUILD</code>	Global flag to set when building the documentation to use the proper PyVista
	Jupyter

## The connection package

### Summary

### Submodules

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

## The backend.py module

### Summary

### Enums

<code>BackendType</code>	Provides an enum holding the available backend types.
<code>ApiVersions</code>	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

<code>DISCOVERY</code>
<code>SPACECLAIM</code>
<code>WINDOWS_SERVICE</code>
<code>LINUX_SERVICE</code>

### 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

<code>V_21</code>
<code>V_22</code>
<code>V_231</code>
<code>V_232</code>
<code>V_241</code>

## 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
ApiVersions.V_241 = 241
```

## Description

Module providing definitions for the backend types.

### The client.py module

#### Summary

#### Classes

<code>GrpcClient</code>	Wraps the gRPC connection for the Geometry service.
-------------------------	---

#### Functions

<code>wait_until_healthy</code>	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.platforminstancemanagement.Instance] = None, docker_instance: beartype.typing.Optional[ansys.geometry.core.connection.docker_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.

### Parameters

#### host

[`str`, default: `DEFAULT_HOST`] Host where the server is running.

#### port

[`Union[str, int]`, default: `DEFAULT_PORT`] Port number where the server is running.

#### channel

[`Channel`, default: `None`] gRPC channel for server communication.

#### remote\_instance

[`ansys.platforminstancemanagement.Instance`, default: `None`] Corresponding remote instance when the Geometry service is launched through `PyPIM`. This instance is deleted when calling the `GrpcClient.close` method.

#### docker\_instance

[`LocalDockerInstance`, default: `None`] Corresponding local Docker instance when the Geometry service is launched using the `launch_docker_modeler()` method. This local Docker instance is deleted when the `GrpcClient.close` method is called.

#### product\_instance

[`ProductInstance`, default: `None`] Corresponding local product instance when the product (Discovery or SpaceClaim) is launched through the `launch_modeler_with_geometry_service()`, `launch_modeler_with_discovery()` or the `launch_modeler_with_spaceclaim()` interface. This instance will be deleted when the `GrpcClient.close` method is called.

#### timeout

[`real`, default: 120] Maximum time to spend trying to make the connection.

#### logging\_level

[`int`, default: `INFO`] Logging level to apply to the client.

#### logging\_file

[`str` or `Path`, default: `None`] File to output the log to, if requested.

#### backend\_type: BackendType, default: None

Type of the backend that PyAnsys Geometry is communicating with. By default, this value is unknown, which results in `None` being the default value.

## Overview

## Methods

<code>close</code>	Close the channel.
<code>target</code>	Get the target of the channel.
<code>get_name</code>	Get the target name of the connection.

## Properties

<code>backend_type</code>	Backend type.
<code>backend_version</code>	Get the current backend version.
<code>multiple_designs_allowed</code>	Flag indicating whether multiple designs are allowed.
<code>channel</code>	Client gRPC channel.
<code>log</code>	Specific instance logger.
<code>is_closed</code>	Flag indicating whether the client connection is closed.
<code>healthy</code>	Flag indicating whether the client channel is healthy.

## Special methods

<code>__repr__</code>	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.backend_version: str`

Get the current backend version.

### Returns

`str`

Backend version in semantic versioning format (that is, Ansys 24R1 SP2 would be 24.1.2).

`property GrpcClient.multiple_designs_allowed: bool`

Flag indicating whether multiple designs are allowed.

## Notes

This method will return `False` if the backend type is `Discovery` or `Linux Service`. Otherwise, it will return `True`.

**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 Docker 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 connection.

## 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

<code>unit_vector_to_grpc_direction</code>	Convert a <code>UnitVector3D</code> class to a unit vector Geometry service gRPC message.
<code>frame_to_grpc_frame</code>	Convert a <code>Frame</code> class to a frame Geometry service gRPC message.
<code>plane_to_grpc_plane</code>	Convert a <code>Plane</code> class to a plane Geometry service gRPC message.
<code>sketch_shapes_to_grpc_geometri</code>	Convert lists of <code>SketchEdge</code> and <code>SketchFace</code> to a <code>GRPCGeometries</code> message.
<code>sketch_edges_to_grpc_geometrie</code>	Convert a list of <code>SketchEdge</code> to a <code>GRPCGeometries</code> gRPC message.
<code>sketch_arc_to_grpc_arc</code>	Convert an <code>Arc</code> class to an arc Geometry service gRPC message.
<code>sketch_ellipse_to_grpc_ellipse</code>	Convert a <code>SketchEllipse</code> class to an ellipse Geometry service gRPC message.
<code>sketch_circle_to_grpc_circle</code>	Convert a <code>SketchCircle</code> class to a circle Geometry service gRPC message.
<code>point3d_to_grpc_point</code>	Convert a <code>Point3D</code> class to a point Geometry service gRPC message.
<code>point2d_to_grpc_point</code>	Convert a <code>Point2D</code> class to a point Geometry service gRPC message.
<code>sketch_polygon_to_grpc_polygon</code>	Convert a <code>Polygon</code> class to a polygon Geometry service gRPC message.
<code>sketch_segment_to_grpc_line</code>	Convert a <code>Segment</code> class to a line Geometry service gRPC message.
<code>tess_to_pd</code>	Convert an <code>ansys.api.geometry.Tessellation</code> to <code>pyvista.PolyData</code> .
<code>grpc_matrix_to_matrix</code>	Convert an <code>ansys.api.geometry.Matrix</code> to a <code>Matrix44</code> .
<code>grpc_frame_to_frame</code>	Convert an <code>ansys.api.geometry.Frame</code> gRPC message to a <code>Frame</code> 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_pb2.  
    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

<code>DEFAULT_HOST</code>	Default for the HOST name.
<code>DEFAULT_PORT</code>	Default for the HOST port.
<code>MAX_MESSAGE_LENGTH</code>	Default for the gRPC maximum message length.
<code>GEOMETRY_SERVICE_DOCKER_IMAGE</code>	Default for the Geometry service Docker image location.
<code>DEFAULT_PIM_CONFIG</code>	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 docker\_instance.py module

### Summary

### Classes

<i>LocalDockerInstance</i>	Instantiates a Geometry service as a local Docker container.
----------------------------	--

### Enums

<i>GeometryContainers</i>	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.

By default, if a container with the Geometry service already exists at the given port, PyAnsys Geometry connects to it. Otherwise, PyAnsys Geometry tries to launch its own service.

### Parameters

#### port

[int, optional] Localhost port to deploy the Geometry service on or 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.

## Overview

## Properties

---

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

---

## Attributes

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

## Static methods

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

## Import detail

```
from ansys.geometry.core.connection.docker_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

---

<code>WINDOWS_LATEST</code>
<code>LINUX_LATEST</code>
<code>WINDOWS_LATEST_UNSTABLE</code>
<code>LINUX_LATEST_UNSTABLE</code>

---

### Import detail

```
from ansys.geometry.core.connection.docker_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

```
docker_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 launcher.py module

### Summary

### Functions

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

## Description

Module for connecting to instances of the Geometry service.

## Module detail

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

Start the Modeler interface for PyAnsys Geometry.

### Parameters

#### mode

[`str`, default: `None`] Mode in which to launch the Modeler service. The default is `None`, in which case the method tries to determine the mode automatically. The possible values are:

- "pypim": Launches the Modeler service remotely using the PIM API.
- "docker": Launches the Modeler service locally using Docker.
- "geometry\_service": Launches the Modeler service locally using the Ansys Geometry Service.
- "spaceclaim": Launches the Modeler service locally using Ansys SpaceClaim.
- "discovery": Launches the Modeler service locally using Ansys Discovery.

#### \*\*kwargs

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

**Returns****`ansys.geometry.core.modeler.Modeler`**

Instance of the Geometry service.

```
launcher.launch_docker_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.docker_instance.GeometryContainer]
                               = 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_docker_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, logs_folder: str = None) → 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.

### logs\_folder

[sets the backend's logs folder path. If nothing is defined,] the backend will use its default path.

## 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,
```

(continues on next page)

(continued from previous page)

```
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, manifest_path: str =  
None, logs_folder: str = None, hidden: bool = False)
```

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.

##### **manifest\_path**

[str, optional] Used to specify a manifest file path for the ApiServerAddin. This way, it is possible to run an ApiServerAddin from a version an older product version.

**logs\_folder**

[sets the backend's logs folder path. If nothing is defined,] the backend will use its default path.

**hidden**

[starts the product hiding its UI. Default is False.]

**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, manifest_path: str = None, logs_folder: str = None, hidden: bool = False)`

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.

**manifest\_path**

[`str`, optional] Used to specify a manifest file path for the ApiServerAddin. This way, it is possible to run an ApiServerAddin from a version an older product version.

**logs\_folder**

[sets the backend's logs folder path. If nothing is defined,] the backend will use its default path.

**hidden**

[starts the product hiding its UI. Default is `False`.]

**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 product\_instance.py module

### Summary

### Classes

<i>ProductInstance</i>	ProductInstance class.
------------------------	------------------------

### Functions

<i>prepare_and_start_backend</i>	Start the requested service locally using the ProductInstance class.
<i>get_available_port</i>	Return an available port to be used.

## Constants

<code>WINDOWS_GEOMETRY_SERVICE_FOLDER</code>	Default Geometry Service's folder name into the unified installer.
<code>DISCOVERY_FOLDER</code>	Default Discovery's folder name into the unified installer.
<code>SPACECLAIM_FOLDER</code>	Default SpaceClaim's folder name into the unified installer.
<code>ADDINS_SUBFOLDER</code>	Default global Addins's folder name into the unified installer.
<code>BACKEND_SUBFOLDER</code>	Default backend's folder name into the <code>ADDINS_SUBFOLDER</code> folder.
<code>MANIFEST_FILENAME</code>	Default backend's addin filename.
<code>GEOMETRY_SERVICE_EXE</code>	The Windows Geometry Service's filename.
<code>DISCOVERY_EXE</code>	The Ansys Discovery's filename.
<code>SPACECLAIM_EXE</code>	The Ansys SpaceClaim's filename.
<code>BACKEND_LOG_LEVEL_VARIABLE</code>	The backend's log level environment variable for local start.
<code>BACKEND_TRACE_VARIABLE</code>	The backend's enable trace environment variable for local start.
<code>BACKEND_HOST_VARIABLE</code>	The backend's ip address environment variable for local start.
<code>BACKEND_PORT_VARIABLE</code>	The backend's port number environment variable for local start.
<code>BACKEND_LOGS_FOLDER_VARIABLE</code>	The backend's logs folder path to be used.
<code>BACKEND_API_VERSION_VARIABLE</code>	The backend's api version environment variable for local start.
<code>BACKEND_SPACECLAIM_OPTIONS</code>	The additional argument for local Ansys Discovery start.
<code>BACKEND_ADDIN_MANIFEST_ARGUMENT</code>	The argument to specify the backend's addin manifest file's path.
<code>BACKEND_SPACECLAIM_HIDDEN</code>	The argument to hide SpaceClaim's UI on the backend.
<code>BACKEND_SPACECLAIM_HIDDEN_ENVVAR_KEY</code>	SpaceClaim hidden backend's environment variable key.
<code>BACKEND_SPACECLAIM_HIDDEN_ENVVAR_VALUE</code>	SpaceClaim hidden backend's environment variable value.
<code>BACKEND_DISCOVERY_HIDDEN</code>	The argument to hide Discovery's UI on the backend.
<code>BACKEND_SPLASH_OFF</code>	The argument to specify the backend's addin manifest file's path.

## ProductInstance

`class ProductInstance(pid: int)`

ProductInstance class.

This class is used as a handle for a local session of Ansys Product's backend: Discovery, Windows Geometry Service or SpaceClaim.

### Parameters

#### pid

[int] The local instance's process identifier. This allows to keep track of the process and close it if need be.

## Overview

## Methods

<code>close</code>	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, manifest_path: str  
            = None, logs_folder: str = None, hidden: bool = False) →  
            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.

**manifest\_path**

[`str`, optional] Used to specify a manifest file path for the ApiServerAddin. This way, it is possible to run an ApiServerAddin from a version an older product version. Only applicable for Ansys Discovery and Ansys SpaceClaim.

**logs\_folder**

[sets the backend's logs folder path. If nothing is defined,] the backend will use its default path.

**hidden**

[starts the product hiding its UI. Default is False.]

**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() → int`

Return an available port to be used.

**Returns**

**int**

The available port.

`product_instance.WINDOWS_GEOMETRY_SERVICE_FOLDER = 'GeometryService'`

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_LOGS_FOLDER_VARIABLE = 'ANS_DSCO_REMOTE_LOGS_FOLDER'
    The backend's logs folder path to be used.

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.

product_instance.BACKEND_SPACECLAIM_HIDDEN = '/Headless=True'
    The argument to hide SpaceClaim's UI on the backend.

    To be used only with Ansys SpaceClaim.

product_instance.BACKEND_SPACECLAIM_HIDDEN_ENVVAR_KEY = 'SPACECLAIM_MODE'
    SpaceClaim hidden backend's environment variable key.

    To be used only with Ansys SpaceClaim.

product_instance.BACKEND_SPACECLAIM_HIDDEN_ENVVAR_VALUE = '2'
    SpaceClaim hidden backend's environment variable value.

    To be used only with Ansys SpaceClaim.

product_instance.BACKEND_DISCOVERY_HIDDEN = '--hidden'
    The argument to hide Discovery's UI on the backend.

    To be used only with Ansys Discovery.

product_instance.BACKEND_SPLASH_OFF = '/Splash=False'
    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

<b>validate</b>	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

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

## The beam.py module

### Summary

### Classes

<code>BeamProfile</code>	Represents a single beam profile organized within the design assembly.
<code>BeamCircularProfile</code>	Represents a single circular beam profile organized within the design assembly.
<code>Beam</code>	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.

This profile synchronizes to a design within a supporting Geometry service instance.

##### Parameters

<code>id</code>	[str] Server-defined ID for the beam profile.
<code>name</code>	[str] User-defined label for the beam profile.

##### Notes

`BeamProfile` objects are expected to be created from the `Design` object. This means that you are not expected to instantiate your own `BeamProfile` object. You should call the specific Design API for the `BeamProfile` desired.

### Overview

### Properties

<code>id</code>	ID of the beam profile.
<code>name</code>	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.

This profile synchronizes to a design within a supporting Geometry service instance.

### Parameters

**id**

[str] Server-defined ID for the beam profile.

**name**

[str] User-defined label for the beam profile.

**radius**

[Distance] Radius of the circle.

**center: Point3D**

3D point representing the center of the circle.

**direction\_x: UnitVector3D**

X-axis direction.

**direction\_y: UnitVector3D**

Y-axis direction.

## Notes

`BeamProfile` objects are expected to be created from the `Design` object. This means that you are not expected to instantiate your own `BeamProfile` object. You should call the specific `Design` API for the `BeamProfile` desired.

## Overview

### Properties

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

### Special methods

<code>__repr__</code>	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.  
     This body synchronizes to a design within a supporting Geometry service instance.

### Parameters

**id**

[`str`] Server-defined ID for the body.

**name**

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

**start**

[`Point3D`] Start of the beam line segment.

**end**

[`Point3D`] End of the beam line segment.

**profile**

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

**parent\_component**

[`Component`] Parent component to nest the new beam under within the design assembly.

## Overview

### Properties

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

### Special methods

<code>__repr__</code>	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**

<code>IBody</code>	Defines the common methods for a body, providing the abstract body interface.
--------------------	---

**Classes**

<code>MasterBody</code>	Represents solids and surfaces organized within the design assembly.
<code>Body</code>	Represents solids and surfaces organized within the design assembly.

**Enums**

<code>MidSurfaceOffsetType</code>	Provides values for mid-surface offsets supported by the Geometry service.
<code>CollisionType</code>	Provides values for collision types between bodies.

## IBody

### class IBody

Bases: `abc.ABC`

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

Both the `MasterBody` class and `Body` class both inherit from the `IBody` class. All child classes must implement all abstract methods.

## Overview

### Abstract methods

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

### Methods

<code>intersect</code>	Intersect two (or more) bodies.
<code>subtract</code>	Subtract two (or more) bodies.
<code>unite</code>	Unite two (or more) 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.rotate(axis_origin: ansys.geometry.core.math.point.Point3D, axis_direction:  
    ansys.geometry.core.math.vector.UnitVector3D, angle:  
    beartype.typing.Union[pint.Quantity, ansys.geometry.core.misc.measurements.Angle,  
    ansys.geometry.core.typing.Real]) → None
```

Rotate the geometry body around the specified axis by a given angle.

### Parameters

#### axis\_origin: Point3D

Origin of the rotational axis.

#### axis\_direction: UnitVector3D

The axis of rotation.

#### angle: Union[~pint.Quantity, Angle, Real]

Angle (magnitude) of the rotation.

### Returns

---

#### None

```
abstract IBody.get_collision(body: Body) → CollisionType
```

Get the collision state between bodies.

### Parameters

#### body: Body

Object that the collision state is checked with.

### Returns

#### CollisionType

Enum that defines the collision state between bodies.

```
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[pv.PolyData, pv.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 `pv.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
```

(continues on next page)

(continued from previous page)

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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Intersect two (or more) 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Subtract two (or more) 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Unite two (or more) 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.

Solids and surfaces synchronize to a design within a supporting Geometry service instance.

**Parameters**

**id**

[`str`] Server-defined ID for the body.

**name**

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

**parent\_component**

[`Component`] Parent component to place the new component under within the design assembly.

**grpc\_client**

[`GrpcClient`] Active supporting geometry service instance for design modeling.

**is\_surface**

[`bool`, default: `False`] Whether the master body is a surface or an 3D object (with volume). The default is `False`, in which case the master body is a surface. When `True`, the master body is a 3D object (with volume).

## Overview

### Abstract methods

<code>imprint_curves</code>	Imprint all specified geometries onto specified faces of the body.
<code>project_curves</code>	Project all specified geometries onto the body.
<code>imprint_projected_curves</code>	Project and imprint specified geometries onto the body.
<code>plot</code>	Plot the body.
<code>intersect</code>	Intersect two (or more) bodies.
<code>subtract</code>	Subtract two (or more) bodies.
<code>unite</code>	Unite two (or more) bodies.

### Methods

<code>reset_tessellation_cache</code>	Decorate <code>MasterBody</code> methods that require a tessellation cache update.
<code>assign_material</code>	Assign a material against the design in the active Geometry service instance.
<code>add_midsurface_thickness</code>	Add a mid-surface thickness to a surface body.
<code>add_midsurface_offset</code>	Add a mid-surface offset to a surface body.
<code>translate</code>	Translate the geometry body in the specified direction by a given distance.
<code>rotate</code>	Rotate the geometry body around the specified axis by a given angle.
<code>get_collision</code>	Get the collision state between bodies.
<code>copy</code>	Create a copy of the body and place it under the specified parent component.
<code>tessellate</code>	Tessellate the body and return the geometry as triangles.

## Properties

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

## Special methods

<code>__repr__</code>	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: UnitVector3D, sketch: Sketch, closest_face: bool, only_one_curve: bool) → List[Face]`

```
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: UnitVector3D, distance: Union[pint.Quantity, Distance, Real]) → None`

```
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.rotate(axis_origin: ansys.geometry.core.math.point.Point3D, axis_direction: ansys.geometry.core.math.vector.UnitVector3D, angle: beartype.typing.Union[pint.Quantity, ansys.geometry.core.misc.measurements.Angle, ansys.geometry.core.typing.Real]) → None`

Rotate the geometry body around the specified axis by a given angle.

#### Parameters

**axis\_origin: Point3D**

Origin of the rotational axis.

**axis\_direction: UnitVector3D**

The axis of rotation.

**angle: Union[~pint.Quantity, Angle, Real]**

Angle (magnitude) of the rotation.

#### Returns

—  
**None**

`MasterBody.get_collision(body: Body) → CollisionType`

Get the collision state between bodies.

#### Parameters

**body: Body**

Object that the collision state is checked with.

#### Returns

**CollisionType**

Enum that defines the collision state between bodies.

`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[pv.vista.PolyData, pv.vista.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** `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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Intersect two (or more) 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Subtract two (or more) 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Unite two (or more) 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_component: ansys.geometry.core.designer.component.Component, template: MasterBody)`

Bases: *IBody*

Represents solids and surfaces organized within the design assembly.

Solids and surfaces synchronize to a design within a supporting Geometry service instance.

**Parameters****id**

[*str*] Server-defined ID for the body.

**name**

[*str*] User-defined label for the body.

**parent\_component**

[*Component*] Parent component to place the new component under within the design assembly.

**template**

[*MasterBody*] Master body that this body is an occurrence of.

## Overview

### Methods

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

### Properties

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

## Special methods

<code>__repr__</code>	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: 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.rotate(axis_origin: ansys.geometry.core.math.point.Point3D, axis_direction: ansys.geometry.core.math.vector.UnitVector3D, angle: beartype.typing.Union[pint.Quantity, ansys.geometry.core.misc.measurements.Angle, ansys.geometry.core.typing.Real]) → None`

Rotate the geometry body around the specified axis by a given angle.

#### Parameters

**axis\_origin: Point3D**

Origin of the rotational axis.

**axis\_direction: UnitVector3D**

The axis of rotation.

**angle: Union[~pint.Quantity, Angle, Real]**

Angle (magnitude) of the rotation.

**Returns**

---

**None**

`Body.get_collision(body: Body) → CollisionType`

Get the collision state between bodies.

**Parameters****body: Body**

Object that the collision state is checked with.

**Returns****CollisionType**

Enum that defines the collision state between bodies.

`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[pv.PolyData, pv.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 `pv.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])
```

(continues on next page)

(continued from previous page)

```
>>> 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Intersect two (or more) 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Subtract two (or more) 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: beartype.typing.Union[Body, beartype.typing.Iterable[Body]]) → None`

Unite two (or more) 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

<code>MIDDLE</code>
<code>TOP</code>
<code>BOTTOM</code>
<code>VARIABLE</code>
<code>CUSTOM</code>

## 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
```

## CollisionType

```
class CollisionType
```

Bases: [enum.Enum](#)

Provides values for collision types between bodies.

## Overview

### Attributes

<i>NONE</i>
<i>TOUCH</i>
<i>INTERSECT</i>
<i>CONTAINED</i>
<i>CONTAINEDTOUCH</i>

## Import detail

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

## Attribute detail

```
CollisionType.NONE = 0  
CollisionType.TOUCH = 1  
CollisionType.INTERSECT = 2  
CollisionType.CONTAINED = 3  
CollisionType.CONTAINEDTOUCH = 4
```

## Description

Provides for managing a body.

### The `component.py` module

#### Summary

#### Classes

<code>Component</code>	Provides for creating and managing a component.
------------------------	---

#### Enums

<code>SharedTopologyType</code>	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.

This class synchronizes to a design within a supporting Geometry service instance.

#### Parameters

##### `name`

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

##### `parent_component`

[`Component` or `None`] Parent component to place the new component under within the design assembly. The default is `None` only when dealing with a `Design` object.

##### `grpc_client`

[`GrpcClient`] Active supporting Geometry service instance for design modeling.

##### `template`

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

##### `preexisting_id`

[`str`, default: `None`] ID of a component pre-existing on the server side to use to create the component on the client-side data model. If an ID is specified, a new component is not created on the server.

**master\_component**

[*MasterComponent*, default: `None`] Master component to use to create a nested component instance instead of creating a new component.

**read\_existing\_comp**

[*bool*, default: `False`] Whether an existing component on the service should be read. This parameter is only valid when connecting to an existing service session. Otherwise, avoid using this optional parameter.

## Overview

### Methods

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

### Properties

<code>id</code>	ID of the component.
<code>name</code>	Name of the component.
<code>components</code>	List of <code>Component</code> objects inside of the component.
<code>bodies</code>	List of <code>Body</code> objects inside of the component.
<code>beams</code>	List of <code>Beam</code> objects inside of the component.
<code>design_points</code>	List of <code>DesignPoint</code> objects inside of the component.
<code>coordinate_systems</code>	List of <code>CoordinateSystem</code> objects inside of the component.
<code>parent_component</code>	Parent of the component.
<code>is_alive</code>	Whether the component is still alive on the server side.
<code>shared_topology</code>	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: beartytype.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: beartytype.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*: *bartype.typing.List[ansys.geometry.core.designer.body.Body]*, *direction*: *ansys.geometry.core.math.vector.UnitVector3D*, *distance*: *bartype.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*: *bartype.typing.List[bartype.typing.Tuple[ansys.geometry.core.math.point.Point3D, ansys.geometry.core.math.point.Point3D]]*, *profile*: *ansys.geometry.core.designer.beam.BeamProfile*) → *bartype.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[pv.PolyData, pv.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 *pv.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

<code>SHARETYPE_NONE</code>
<code>SHARETYPE_SHARE</code>
<code>SHARETYPE_MERGE</code>
<code>SHARETYPE_GROUPS</code>

### 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

<code>CoordinateSystem</code>	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.

This class synchronizes to a design within a supporting Geometry service instance.

#### Parameters

##### `name`

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

##### `frame`

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

##### `parent_component`

[`Component`, default: `Component`] Parent component the coordinate system is assigned against.

##### `grpc_client`

[`GrpcClient`] Active supporting Geometry service instance for design modeling.

### Overview

### Properties

<code>id</code>	ID of the coordinate system.
<code>name</code>	Name of the coordinate system.
<code>frame</code>	Frame of the coordinate system.
<code>parent_component</code>	Parent component of the coordinate system.
<code>is_alive</code>	Flag indicating if coordinate system is still alive on the server side.

## Special methods

<code>__repr__</code>	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

<code>Design</code>	Provides for organizing geometry assemblies.
---------------------	--

## Enums

<code>DesignFFormat</code>	Provides supported file formats that can be downloaded for designs.
----------------------------	---

## Design

`class Design(name: str, modeler: ansys.geometry.core.modeler.Modeler, read_existing_design: bool = False)`

Bases: `ansys.geometry.core.designer.component.Component`

Provides for organizing geometry assemblies.

This class synchronizes to a supporting Geometry service instance.

### Parameters

#### `name`

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

#### `grpc_client`

[`GrpcClient`] Active supporting Geometry service instance for design modeling.

#### `read_existing_design`

[`bool`, default: `False`] Whether an existing design on the service should be read. This parameter is only valid when connecting to an existing service session. Otherwise, avoid using this optional parameter.

## Overview

## Methods

<code>add_material</code>	Add a material to the design.
<code>save</code>	Save a design to disk on the active Geometry server instance.
<code>download</code>	Export and download the design from the active Geometry server instance.
<code>create_named_selection</code>	Create a named selection on the active Geometry server instance.
<code>delete_named_selection</code>	Delete a named selection on the active Geometry server instance.
<code>delete_component</code>	Delete a component (itself or its children).
<code>set_shared_topology</code>	Set the shared topology to apply to the component.
<code>add_beam_circular_profile</code>	Add a new beam circular profile under the design for the creating beams.
<code>add_midsurface_thickness</code>	Add a mid-surface thickness to a list of bodies.
<code>add_midsurface_offset</code>	Add a mid-surface offset type to a list of bodies.
<code>delete_beam_profile</code>	Remove a beam profile on the active geometry server instance.

## Properties

<code>design_id</code>	The design's object unique id.
<code>materials</code>	List of materials available for the design.
<code>named_selections</code>	List of named selections available for the design.
<code>beam_profiles</code>	List of beam profile available for the design.
<code>is_active</code>	Whether the design is currently active.

## Special methods

<code>__repr__</code>	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.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.

`property Design.is_active: bool`

Whether the design is currently active.

## 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.DesignPoint]] = 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:`

`bartype.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: bartype.typing.Union[pint.Quantity,`  
`ansys.geometry.core.misc.measurements.Distance], center:`  
`bartype.typing.Union[numpy.ndarray,`  
`ansys.geometry.core.typing.RealSequence,`  
`ansys.geometry.core.math.point.Point3D] = ZERO_POINT3D,`  
`direction_x: bartype.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: bartype.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

<i>SCDOCX</i>
<i>PARASOLID_TEXT</i>
<i>PARASOLID_BIN</i>
<i>FMD</i>
<i>STEP</i>
<i>IGES</i>
<i>PMDB</i>
<i>INVALID</i>

### 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

<code>DesignPoint</code>	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.

##### Parameters

###### `id`

[`str`] Server-defined ID for the design points.

###### `name`

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

###### `points`

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

###### `parent_component`

[`Component`] Parent component to place the new design point under within the design assembly.

#### Overview

#### Properties

<code>id</code>	ID of the design point.
<code>name</code>	Name of the design point.
<code>value</code>	Value of the design point.
<code>parent_component</code>	Component node that the design point is under.

## Special methods

<code>__repr__</code>	Represent the design points as a string.
-----------------------	--

## Import detail

```
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

<code>Edge</code>	Represents a single edge of a body within the design assembly.
-------------------	--

## Enums

<code>CurveType</code>	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.

This class synchronizes to a design within a supporting Geometry service instance.

### Parameters

<b>id</b>	[str] Server-defined ID for the body.
<b>curve_type</b>	[CurveType] Type of curve that the edge forms.
<b>body</b>	[Body] Parent body that the edge constructs.
<b>grpc_client</b>	[GrpcClient] Active supporting Geometry service instance for design modeling.

## Overview

### Properties

<code>id</code>	ID of the edge.
<code>length</code>	Calculated length of the edge.
<code>curve_type</code>	Curve type of the edge.
<code>faces</code>	Faces that contain the edge.
<code>start_point</code>	Edge start point.
<code>end_point</code>	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

<code>CURVETYPE_UNKNOWN</code>
<code>CURVETYPE_LINE</code>
<code>CURVETYPE_CIRCLE</code>
<code>CURVETYPE_ELLIPSE</code>
<code>CURVETYPE_NURBS</code>
<code>CURVETYPE_PROCEDURAL</code>

## 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

<code>FaceLoop</code>	Provides an internal class holding the face loops defined on the server side.
<code>Face</code>	Represents a single face of a body within the design assembly.

### Enums

<code>SurfaceType</code>	Provides values for the surface types supported by the Geometry service.
<code>FaceLoopType</code>	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:
               beartytype.typing.List[ansys.geometry.core.designer.edge.Edge])
```

Provides an internal class holding the face loops defined on the server side.

### Parameters

#### `type`

[`FaceLoopType`] Type of loop.

#### `length`

[`Quantity`] Length of the loop.

#### `min_bbox`

[`Point3D`] Minimum point of the bounding box containing the loop.

**max\_bbox**

[*Point3D*] Maximum point of the bounding box containing the loop.

**edges**

[*List[Edge]*] Edges contained in the loop.

**Notes**

This class is to be used only when parsing server side results. It is not intended to be instantiated by a user.

**Overview****Properties**

<b>type</b>	Type of the loop.
<b>length</b>	Length of the loop.
<b>min_bbox</b>	Minimum point of the bounding box containing the loop.
<b>max_bbox</b>	Maximum point of the bounding box containing the loop.
<b>edges</b>	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.

This class synchronizes to a design within a supporting Geometry service instance.

### Parameters

#### **id**

[str] Server-defined ID for the body.

#### **surface\_type**

[SurfaceType] Type of surface that the face forms.

#### **body**

[Body] Parent body that the face constructs.

#### **grpc\_client**

[GrpcClient] Active supporting Geometry service instance for design modeling.

## Overview

### Methods

<code>face_normal</code>	Get the normal direction to the face evaluated at certain UV coordinates.
<code>face_point</code>	Get a point of the face evaluated at certain UV coordinates.

### Properties

<code>id</code>	Face ID.
<code>body</code>	Body that the face belongs to.
<code>area</code>	Calculated area of the face.
<code>surface_type</code>	Surface type of the face.
<code>edges</code>	List of all edges of the face.
<code>loops</code>	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

<code>SURFACETYPE_UNKNOWN</code>
<code>SURFACETYPE_PLANE</code>
<code>SURFACETYPE_CYLINDER</code>
<code>SURFACETYPE_CONE</code>
<code>SURFACETYPE_TORUS</code>
<code>SURFACETYPE_SPHERE</code>
<code>SURFACETYPE_NURBS</code>
<code>SURFACETYPE_PROCEDURAL</code>

### 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

<i>INNER_LOOP</i>
<i>OUTER_LOOP</i>

## 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

<i>Part</i>	Represents a part master.
<i>MasterComponent</i>	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.

This class should not be accessed by users. The `Part` class holds fundamental data of an assembly.

### Parameters

#### `id`

[`str`] Unique identifier for the part.

#### `name`

[`str`] Name of the part.

#### `components`

[`List[MasterComponent]`] List of `MasterComponent` children that the part contains.

#### `bodies`

[`List[MasterBody]`] List of `MasterBody` children that the part contains. These are master bodies.

## Overview

## Properties

<code>id</code>	ID of the part.
<code>name</code>	Name of the part.
<code>components</code>	<code>MasterComponent</code> children that the part contains.
<code>bodies</code>	<code>MasterBody</code> children that the part contains.

## Special methods

<code>__repr__</code>	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.

### Parameters

<b>id</b>	[str] Unique identifier for the transformed part.
<b>name</b>	[str] Name of the transformed part.
<b>part</b>	[Part] Reference to the transformed part's master part.
<b>transform</b>	[Matrix44] 4x4 transformation matrix from the master part.

### Notes

This class should not be accessed by users. It holds the fundamental data of an assembly. Master components wrap parts by adding a transform matrix.

## Overview

### Properties

<code>id</code>	ID of the transformed part.
<code>name</code>	Name of the transformed part.
<code>occurrences</code>	List of all occurrences of the component.
<code>part</code>	Master part of the transformed part.
<code>transform</code>	4x4 transformation matrix from the master part.

### Special methods

<code>__repr__</code>	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

<code>NamedSelection</code>	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.

This class synchronizes to a design within a supporting Geometry service instance.

A named selection organizes one or more design entities together for common actions against the entire group.

#### Parameters

##### `name`

[`str`] User-defined name for the named selection.

##### `grpc_client`

[`GrpcClient`] Active supporting Geometry service instance for design modeling.

##### `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[DesignPoints]`, default: `None`] All design points to include in the named selection.

## Overview

### Properties

<i>id</i>	ID of the named selection.
<i>name</i>	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

<code>material</code>	Provides the data structure for material and for adding a material property.
<code>property</code>	Provides the <code>MaterialProperty</code> class.

## The material.py module

### Summary

### Classes

<b>Material</b>	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.

##### Parameters

**name:** str

Material name.

**density:** ~pint.Quantity

Material density.

**additional\_properties:** Sequence[MaterialProperty], **default:** None

Additional material properties.

### Overview

### Methods

<b>add_property</b>	Add a material property to the Material class.
---------------------	--

### Properties

<b>properties</b>	Dictionary of the material property type and material properties.
<b>name</b>	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

<code>MaterialProperty</code>	Provides the data structure for a material property.
-------------------------------	--

### Enums

<code>MaterialPropertyType</code>	Provides an enum holding the possible values for <code>MaterialProperty</code> objects.
-----------------------------------	---

## MaterialProperty

```
class MaterialProperty(type: beartype.typing.Union[Material.PropertyType, str], name: str, quantity: beartype.typing.Union[pint.Quantity, ansys.geometry.core.typing.Real])
```

Provides the data structure for a material property.

### Parameters

#### type

[Union[Material.PropertyType, str]] Type of the material property. If the type is a string, it must be a valid material property type - though it might not be supported by the Material.PropertyType enum.

#### name: str

Material property name.

#### quantity: Union[-pint.Quantity, Real]

Value and unit in case of a supported Quantity. If the type is not supported, it must be a Real value (float or integer).

## Overview

### Properties

<code>type</code>	Material property ID.
<code>name</code>	Material property name.
<code>quantity</code>	Material property quantity and unit.

### Import detail

```
from ansys.geometry.core.materials.property import MaterialProperty
```

### Property detail

```
property MaterialProperty.type: beartype.typing.Union[Material.PropertyType, str]
```

Material property ID.

If the type is not supported, it will be a string.

```
property MaterialProperty.name: str
```

Material property name.

```
property MaterialProperty.quantity: beartype.typing.Union[pint.Quantity, ansys.geometry.core.typing.Real]
```

Material property quantity and unit.

If the type is not supported, it will be a Real value (float or integer).

## MaterialPropertyType

**class MaterialPropertyType**

Bases: [enum.Enum](#)

Provides an enum holding the possible values for MaterialProperty objects.

### Overview

### Methods

<i>from_id</i>	Return the MaterialPropertyType value from the service representation.
----------------	--

### Attributes

DENSITY
ELASTIC_MODULUS
POISSON_RATIO
SHEAR_MODULUS
SPECIFIC_HEAT
TENSILE_STRENGTH
THERMAL_CONDUCTIVITY

### Import detail

<code>from ansys.geometry.core.materials.property import MaterialPropertyType</code>
--

### 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

`Material.PropertyType.from_id()` → `Material.PropertyType`

Return the `Material.PropertyType` value from the service representation.

### Parameters

`id`

[`str`] Geometry Service string representation of a property type.

### Returns

`Material.PropertyType`

Common name for property type.

## Description

Provides the `MaterialProperty` class.

## Description

PyAnsys Geometry materials subpackage.

## The `math` package

### Summary

### Submodules

<code>bbox</code>	Provides for managing a bounding box.
<code>constants</code>	Provides mathematical constants.
<code>frame</code>	Provides for managing a frame.
<code>matrix</code>	Provides matrix primitive representations.
<code>plane</code>	Provides primitive representation of a 2D plane in 3D space.
<code>point</code>	Provides geometry primitive representation for 2D and 3D points.
<code>vector</code>	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.

### Parameters

#### x\_min

[Real] Minimum value for the x-dimensional bounds.

#### x\_max

[Real] Maximum value for the x-dimensional bounds.

#### y\_min

[Real] Minimum value for the y-dimensional bounds.

#### y\_max

[Real] Maximum value for the y-dimensional bounds.

## Overview

### Methods

<code>add_point</code>	Extend the ranges of the bounding box to include a point.
<code>add_point_components</code>	Extend the ranges of the bounding box to include the X and Y values.
<code>add_points</code>	Extend the ranges of the bounding box to include given points.
<code>contains_point</code>	Evaluate whether a provided point lies within the X and Y ranges of the bounds.
<code>contains_point_components</code>	Check if point components are within current X and Y ranges of the bounds.

### Properties

<code>x_min</code>	Minimum value of X-dimensional bounds.
<code>x_max</code>	Maximum value of the X-dimensional bounds.
<code>y_min</code>	Minimum value of Y-dimensional bounds.
<code>y_max</code>	Maximum value of Y-dimensional bounds.

### Special methods

<code>__eq__</code>	Equals operator for the BoundingBox2D class.
<code>__ne__</code>	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

<code>DEFAULT_POINT3D</code>	Default value for a 3D point.
<code>DEFAULT_POINT2D</code>	Default value for a 2D point.
<code>IDENTITY_MATRIX33</code>	Identity for a <code>Matrix33</code> object.
<code>IDENTITY_MATRIX44</code>	Identity for a <code>Matrix44</code> object.
<code>UNITVECTOR3D_X</code>	Default 3D unit vector in the Cartesian traditional X direction.
<code>UNITVECTOR3D_Y</code>	Default 3D unit vector in the Cartesian traditional Y direction.
<code>UNITVECTOR3D_Z</code>	Default 3D unit vector in the Cartesian traditional Z direction.
<code>UNITVECTOR2D_X</code>	Default 2D unit vector in the Cartesian traditional X direction.
<code>UNITVECTOR2D_Y</code>	Default 2D unit vector in the Cartesian traditional Y direction.
<code>ZERO_VECTOR3D</code>	Zero-valued <code>Vector3D</code> object.
<code>ZERO_VECTOR2D</code>	Zero-valued <code>Vector2D</code> object.
<code>ZERO_POINT3D</code>	Zero-valued <code>Point3D</code> object.
<code>ZERO_POINT2D</code>	Zero-valued <code>Point2D</code> 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

<b>Frame</b>	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).

#### Parameters

##### origin

[Union[ndarray, RealSequence, Point3D], default: ZERO\_POINT3D] Centered origin of the frame. The default is ZERO\_POINT3D, which is the Cartesian origin.

##### direction\_x

[Union[ndarray, RealSequence, UnitVector3D, Vector3D], default: UNITVECTOR3D\_X] X-axis direction.

**direction\_y**  
[Union[ndarray, RealSequence, UnitVector3D, Vector3D], default: UNITVECTOR3D\_Y] Y-axis direction.

## Overview

### Methods

<code>transform_point2d_local_to_global</code>	Transform a 2D point to a global 3D point.
--	--

### Properties

<code>origin</code>	Origin of the frame.
<code>direction_x</code>	X-axis direction of the frame.
<code>direction_y</code>	Y-axis direction of the frame.
<code>direction_z</code>	Z-axis direction of the frame.
<code>global_to_local_rotation</code>	Global to local space transformation matrix.
<code>local_to_global_rotation</code>	Local to global space transformation matrix.
<code>transformation_matrix</code>	Full 4x4 transformation matrix.

### Special methods

<code>__eq__</code>	Equals operator for the Frame class.
<code>__ne__</code>	Not equals operator for the Frame class.

### Import detail

<code>from ansys.geometry.core.math.frame import Frame</code>
---

### 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

<code>Matrix</code>	Provides matrix primitive representation.
<code>Matrix33</code>	Provides 3x3 matrix primitive representation.
<code>Matrix44</code>	Provides 4x4 matrix primitive representation.

### Constants

<code>DEFAULT_MATRIX33</code>	Default value of the 3x3 identity matrix for the <code>Matrix33</code> class.
<code>DEFAULT_MATRIX44</code>	Default value of the 4x4 identity matrix for the <code>Matrix44</code> class.

## Matrix

`class Matrix(shape, dtype=float, buffer=None, offset=0, strides=None, order=None)`

Bases: `numpy.ndarray`

Provides matrix primitive representation.

### Parameters

#### input

[Union[`ndarray`, `RealSequence`]] Matrix arguments as a `np.ndarray` class.

### Overview

### Methods

<code>determinant</code>	Get the determinant of the matrix.
<code>inverse</code>	Provide the inverse of the matrix.

## Special methods

<code>__mul__</code>	Get the multiplication of the matrix.
<code>__eq__</code>	Equals operator for the <code>Matrix</code> class.
<code>__ne__</code>	Not equals operator for the <code>Matrix</code> 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.

### Parameters

#### input

[Union[`ndarray`, `RealSequence`, `Matrix`], default: `DEFAULT_MATRIX33`] Matrix arguments as a `np.ndarray` class.

## 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.

### Parameters

#### input

[Union[`ndarray`, `RealSequence`, `Matrix`], default: `DEFAULT_MATRIX44`] Matrix arguments as a `np.ndarray` class.

## 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

<code>Plane</code>	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.

### Parameters

#### `origin`

[Union[ndarray, RealSequence, Point3D], default: ZERO\_POINT3D] Centered origin of the frame. The default is ZERO\_POINT3D, which is the Cartesian origin.

#### `direction_x`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D], default: UNITVECTOR3D\_X] X-axis direction.

#### `direction_y`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D], default: UNITVECTOR3D\_Y] Y-axis direction.

## Overview

### Methods

<code>is_point_contained</code>	Check if a 3D point is contained in the plane.
---------------------------------	--

### Properties

<code>normal</code>	Calculate the normal vector of the plane.
---------------------	---

### Special methods

<code>__eq__</code>	Equals operator for the Plane class.
<code>__ne__</code>	Not equals operator for the Plane class.

### Import detail

<code>from ansys.geometry.core.math.plane import Plane</code>
---

### Property detail

`property Plane.normal: UnitVector3D`

Calculate the normal vector of the plane.

#### Returns

##### `UnitVector3D`

Normal vector of the 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

<code>Point2D</code>	Provides geometry primitive representation for a 2D point.
<code>Point3D</code>	Provides geometry primitive representation for a 3D point.

### Constants

<code>DEFAULT_POINT2D_VALUES</code>	Default values for a 2D point.
<code>DEFAULT_POINT3D_VALUES</code>	Default values for a 3D point.
<code>BASE_UNIT_LENGTH</code>	Default value for the length of the base unit.

## Point2D

```
class Point2D(input: beartype.typing.Union[numumpy.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.

### Parameters

#### input

[Union[`ndarray`, `RealSequence`], default: `DEFAULT_POINT2D_VALUES`] Direction arguments, either as a `numpy.ndarray` class or as a `RealSequence`.

#### unit

[`Unit`, default: `DEFAULT_UNITS.LENGTH`] Units for defining 2D point values.

## Overview

### Methods

<code>unit</code>	Get the unit of the object.
<code>base_unit</code>	Get the base unit of the object.

### Properties

<code>x</code>	X plane component value.
<code>y</code>	Y plane component value.

### Special methods

<code>__eq__</code>	Equals operator for the <code>Point2D</code> class.
<code>__ne__</code>	Not equals operator for the <code>Point2D</code> class.
<code>__add__</code>	Add operation for the <code>Point2D</code> class.
<code>__sub__</code>	Subtraction operation for the <code>Point2D</code> 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[numumpy.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.

### Parameters

#### `input`

[Union[`ndarray`, `RealSequence`], default: `DEFAULT_POINT3D_VALUES`] Direction arguments, either as a `numpy.ndarray` class or as a `RealSequence`.

#### `unit`

[`Unit`, default: `DEFAULT_UNITS.LENGTH`] Units for defining 3D point values.

## Overview

### Methods

<code>unit</code>	Get the unit of the object.
<code>base_unit</code>	Get the base unit of the object.
<code>transform</code>	Transform the 3D point with a transformation matrix.

### Properties

<code>x</code>	X plane component value.
<code>y</code>	Y plane component value.
<code>z</code>	Z plane component value.

### Special methods

<code>__eq__</code>	Equals operator for the Point3D class.
<code>__ne__</code>	Not equals operator for the Point3D class.
<code>__add__</code>	Add operation for the Point3D class.
<code>__sub__</code>	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

<code>Vector3D</code>	Provides for managing and creating a 3D vector.
<code>Vector2D</code>	Provides for creating and managing a 2D vector.
<code>UnitVector3D</code>	Provides for creating and managing a 3D unit vector.
<code>UnitVector2D</code>	Provides for creating and managing a 2D 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.

#### Parameters

<code>input</code>	[Union[ <code>ndarray</code> , <code>RealSequence</code> ]] 3D <code>numpy.ndarray</code> class with shape(X,).
--------------------	---

### Overview

### Constructors

<code>from_points</code>	Create a 3D vector from two distinct 3D points.
--------------------------	---

### Methods

<code>is_perpendicular_to</code>	Check if this vector and another vector are perpendicular.
<code>is_parallel_to</code>	Check if this vector and another vector are parallel.
<code>is_opposite</code>	Check if this vector and another vector are opposite.
<code>normalize</code>	Return a normalized version of the 3D vector.
<code>transform</code>	Transform the 3D vector3D with a transformation matrix.
<code>get_angle_between</code>	Get the angle between this 3D vector and another 3D vector.
<code>cross</code>	Get the cross product of Vector3D objects.

## Properties

<code>x</code>	X coordinate of the Vector3D class.
<code>y</code>	Y coordinate of the Vector3D class.
<code>z</code>	Z coordinate of the Vector3D class.
<code>norm</code>	Norm of the vector.
<code>magnitude</code>	Norm of the vector.
<code>is_zero</code>	Check if all components of the 3D vector are zero.

## Special methods

<code>__eq__</code>	Equals operator for the Vector3D class.
<code>__ne__</code>	Not equals operator for the Vector3D class.
<code>__mul__</code>	Overload * operator with dot product.
<code>__mod__</code>	Overload % operator with cross product.
<code>__add__</code>	Addition operation overload for 3D vectors.
<code>__sub__</code>	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[numumpy.ndarray, ansys.geometry.core.typing.RealSequence, ansys.geometry.core.math.point.Point3D], point_b: beartype.typing.Union[numumpy.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.

### Parameters

**input**  
[`Union[ndarray, RealSequence]`] 2D `numpy.ndarray` class with `shape(X, )`.

## Overview

### Constructors

<code>from_points</code>	Create a 2D vector from two distinct 2D points.
--------------------------	---

### Methods

<code>cross</code>	Return the cross product of Vector2D objects.
<code>is_perpendicular_to</code>	Check if this 2D vector and another 2D vector are perpendicular.
<code>is_parallel_to</code>	Check if this vector and another vector are parallel.
<code>is_opposite</code>	Check if this vector and another vector are opposite.
<code>normalize</code>	Return a normalized version of the 2D vector.
<code>get_angle_between</code>	Get the angle between this 2D vector and another 2D vector.

### Properties

<code>x</code>	X coordinate of the 2D vector.
<code>y</code>	Y coordinate of the 2D vector.
<code>norm</code>	Norm of the 2D vector.
<code>magnitude</code>	Norm of the 2D vector.
<code>is_zero</code>	Check if values for all components of the 2D vector are zero.

### Special methods

<code>__eq__</code>	Equals operator for the Vector2D class.
<code>__ne__</code>	Not equals operator for the Vector2D class.
<code>__mul__</code>	Overload * operator with dot product.
<code>__add__</code>	Addition operation overload for 2D vectors.
<code>__sub__</code>	Subtraction operation overload for 2D vectors.
<code>__mod__</code>	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.

### Parameters

`input`  
[`ndarray`, `Vector3D`]

- 1D `numpy.ndarray` class with `shape(X,)`
- `Vector3D`

## 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[numumpy.ndarray,
                                         ansys.geometry.core.typing.RealSequence,
                                         ansys.geometry.core.math.point.Point3D], point_b:
                                         beartype.typing.Union[numumpy.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.

#### Parameters

**input**  
[ndarray, Vector2D]

- 1D numpy.ndarray class with shape(X,)
- Vector2D

## 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

<code>accuracy</code>	Provides for evaluating decimal precision.
<code>auxiliary</code>	Auxiliary functions for the PyAnsys Geometry library.
<code>checks</code>	Provides functions for performing common checks.
<code>measurements</code>	Provides various measurement-related classes.
<code>options</code>	Provides various option classes.
<code>units</code>	Provides for handling units homogeneously throughout PyAnsys Geometry.

## The `accuracy.py` module

### Summary

### Classes

<code>Accuracy</code>	Provides decimal precision evaluations for actions such as equivalency.
-----------------------	---

### Constants

<code>LENGTH_ACCURACY</code>	Constant for decimal accuracy in length comparisons.
<code>ANGLE_ACCURACY</code>	Constant for decimal accuracy in angle comparisons.

## Accuracy

### `class Accuracy`

Provides decimal precision evaluations for actions such as equivalency.

### Overview

## Methods

<code>length_is_equal</code>	Check if the comparison length is equal to the reference length.
<code>length_is_greater_than_or_equal</code>	Check if the comparison length is greater than the reference length.
<code>length_is_less_than_or_equal</code>	Check if the comparison length is less than or equal to the reference length.
<code>length_is_zero</code>	Check if the length is within the length accuracy of exact zero.
<code>length_is_negative</code>	Check if the length is below a negative length accuracy.
<code>length_is_positive</code>	Check if the length is above a positive length accuracy.
<code>angle_is_zero</code>	Check if the length is within the angle accuracy of exact zero.
<code>angle_is_negative</code>	Check if the angle is below a negative angle accuracy.
<code>angle_is_positive</code>	Check if the angle is above a positive angle accuracy.
<code>is_within_tolerance</code>	Check if two values (a and b) are inside a relative and absolute tolerance.

## Import detail

```
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 auxiliary.py module

### Summary

### Functions

<code>get_design_from_component</code>	Get the Design of the given Component object.
<code>get_design_from_body</code>	Get the Design of the given Body object.
<code>get_design_from_face</code>	Get the Design of the given Face object.
<code>get_design_from_edge</code>	Get the Design of the given Edge object.
<code>get_bodies_from_ids</code>	Find the Body objects inside a Design from its ids.
<code>get_faces_from_ids</code>	Find the Face objects inside a Design from its ids.
<code>get_edges_from_ids</code>	Find the Edge objects inside a Design from its ids.

### Description

Auxiliary functions for the PyAnsys Geometry library.

### Module detail

`auxiliary.get_design_from_component(component: ansys.geometry.core.designer.component.Component) → ansys.geometry.core.designer.design.Design`

Get the Design of the given Component object.

#### Parameters

##### `component`

[*Component*] The component object for which to find the Design.

#### Returns

##### `Design`

The Design of the provided component object.

`auxiliary.get_design_from_body(body: ansys.geometry.core.designer.body.Body) → ansys.geometry.core.designer.design.Design`

Get the Design of the given Body object.

#### Parameters

##### `body`

[*Body*] The body object for which to find the Design.

#### Returns

##### `Design`

The Design of the provided body object.

`auxiliary.get_design_from_face(face: ansys.geometry.core.designer.face.Face) → ansys.geometry.core.designer.design.Design`

Get the Design of the given Face object.

#### Parameters

**face**

[Face] The face object for which to find the Design.

**Returns****Design**

The Design of the provided face object.

```
auxiliary.get_design_from_edge(edge: ansys.geometry.core.designer.edge.Edge) →  
    ansys.geometry.core.designer.design.Design
```

Get the Design of the given Edge object.

**Parameters****edge**

[Edge] The edge object for which to find the Design.

**Returns****Design**

The Design of the provided edge object.

```
auxiliary.get_bodies_from_ids(design: ansys.geometry.core.designer.design.Design, body_ids:  
    beartype.typing.List[str]) →  
    beartype.typing.List[ansys.geometry.core.designer.body.Body]
```

Find the Body objects inside a Design from its ids.

**Parameters****design**

[Design] Parent design for the faces.

**body\_ids**

[List[str]] List of body ids.

**Returns****List[Body]**

List of Body objects.

**Notes**

This method takes a design and body ids, and gets their corresponding Body object.

```
auxiliary.get_faces_from_ids(design: ansys.geometry.core.designer.design.Design, face_ids:  
    beartype.typing.List[str]) →  
    beartype.typing.List[ansys.geometry.core.designer.face.Face]
```

Find the Face objects inside a Design from its ids.

**Parameters****design**

[Design] Parent design for the faces.

**face\_ids**

[List[str]] List of face ids.

**Returns****List[Face]**

List of Face objects.

## Notes

This method takes a design and face ids, and gets their corresponding Face object.

```
auxiliary.get_edges_from_ids(design: ansys.geometry.core.designer.design.Design, edge_ids:  
    beartype.typing.List[str]) →  
    beartype.typing.List[ansys.geometry.core.designer.edge.Edge]
```

Find the Edge objects inside a Design from its ids.

### Parameters

#### design

[Design] Parent design for the edges.

#### edge\_ids

[List[str]] List of edge ids.

### Returns

#### List[Edge]

List of Edge objects.

## Notes

This method takes a design and edge ids, and gets their corresponding Edge objects.

## The checks.py module

### Summary

### Functions

<code>ensure_design_is_active</code>	Make sure that the design is active before executing a method.
<code>check_is_float_int</code>	Check if a parameter has a float or integer value.
<code>check_ndarray_is_float_int</code>	Check if a <code>numpy.ndarray</code> has float or integer values.
<code>check_ndarray_is_not_none</code>	Check if a <code>numpy.ndarray</code> has all None values.
<code>check_ndarray_is_all_nan</code>	Check if a <code>numpy.ndarray</code> is all nan-valued.
<code>check_ndarray_is_non_zero</code>	Check if a <code>numpy.ndarray</code> is zero-valued.
<code>check_pint_unit_compatibility</code>	Check if input for <code>pint.Unit</code> is compatible with the expected input.
<code>check_type_equivalence</code>	Check if an input object is of the same class as an expected object.
<code>check_type</code>	Check if an input object is of the same type as expected types.
<code>check_type_all_elements_in_iterable</code>	Check if all elements in an iterable are of the same type as expected types.
<code>min_backend_version</code>	Compare a method's minimum required version to the current backend version.

## Description

Provides functions for performing common checks.

## Module detail

### `checks.ensure_design_is_active(method)`

Make sure that the design is active before executing a method.

This function is necessary to be called whenever we do any operation on the design. If we are just accessing information of the class, it is not necessary to call this.

### `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.

`checks.check_type_all_elements_in_iterable(input: beartype.typing.Iterable, expected_type: beartype.typing.Union[type, beartype.typing.Tuple[type, beartype.typing.Any]]) → None`

Check if all elements in an iterable are of the same type as expected types.

**Parameters****input**

[`Iterable`] Input iterable.

**expected\_type**

[`Union[type, Tuple[type, ...]]`] One or more types to compare the input object against.

**Raises****`TypeError`**

If one of the elements in the iterable does not match the one or more expected types.

`checks.min_backend_version(major: int, minor: int, service_pack: int)`

Compare a method's minimum required version to the current backend version.

**Parameters****major**

[`int`] Minimum major version required by the method.

**minor**

[`int`] Minimum minor version required by the method.

**service\_pack**

[`int`] Minimum service pack version required by the method.

**Raises****`GeometryRuntimeError`**

If the method version is higher than the backend version.

**`GeometryRuntimeError`**

If the client is not available.

## The `measurements.py` module

### Summary

### Classes

<code>SingletonMeta</code>	Provides a thread-safe implementation of a singleton design pattern.
<code>DefaultUnitsClass</code>	Provides default units for the PyAnsys Geometry singleton design pattern.
<code>Measurement</code>	Provides the <code>PhysicalQuantity</code> subclass for holding a measurement.
<code>Distance</code>	Provides the <code>Measurement</code> subclass for holding a distance.
<code>Angle</code>	Provides the <code>Measurement</code> subclass for holding an angle.

### Constants

<code>DEFAULT_UNITS</code>	PyAnsys Geometry default units object.
----------------------------	--

### `SingletonMeta`

```
class SingletonMeta
```

Bases: `type`

Provides a thread-safe implementation of a singleton design pattern.

### Overview

### Special methods

<code>__call__</code>	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

<code>LENGTH</code>	Default length unit for PyAnsys Geometry.
<code>ANGLE</code>	Default angle unit for PyAnsys Geometry.
<code>SERVER_LENGTH</code>	Default length unit for supporting Geometry services for gRPC messages.
<code>SERVER_AREA</code>	Default area unit for supporting Geometry services for gRPC messages.
<code>SERVER_VOLUME</code>	Default volume unit for supporting Geometry services for gRPC messages.
<code>SERVER_ANGLE</code>	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.

### Parameters

#### value

[Union[Real, Quantity]] Value of the measurement.

#### unit

[Unit] Units for the measurement.

#### dimensions

[Unit] Units for extracting the dimensions of the measurement. If ~`pint.Unit.meter` is given, the dimension extracted is [length].

## Overview

## Properties

<code>value</code>	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.

### Parameters

**value**  
[Union[Real, Quantity]] Value of the distance.

**unit**  
[Unit, default: DEFAULT\_UNITS.LENGTH] Units for the 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.

### Parameters

**value**  
[Union[Real, Quantity]] Value of the angle.

**unit**  
[`Unit`, default: `DEFAULT_UNITS.ANGLE`] Units for the distance.

## 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

<i>ImportOptions</i>	Import options when opening a file.
----------------------	-------------------------------------

## ImportOptions

```
class ImportOptions
```

Import options when opening a file.

### Parameters

**cleanup\_bodies**  
[bool = `False`] Simplify geometry and clean up topology.

```
import_coordinate_systems
    [bool = False] Import coordinate systems.

import_curves
    [bool = False] Import curves.

import_hidden_components_and_geometry
    [bool = False] Import hidden components and geometry.

import_names
    [bool = False] Import names of bodies and curves.

import_planes
    [bool = False] Import planes.

import_points
    [bool = False] Import points.
```

## 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

<code>PhysicalQuantity</code>	Provides the base class for handling units throughout PyAnsys Geometry.
-------------------------------	---

#### Constants

<code>UNITS</code>	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.

### Parameters

#### `unit`

[`Unit`] Units for the class.

#### `expected_dimensions`

[`Unit`, default: `None`] Units for the dimensionality of the physical quantity.

## Overview

### Properties

<i>unit</i>	Unit of the object.
<i>base_unit</i>	Base unit of the object.

### Import detail

```
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

<code>widgets</code>	Submodule providing widgets for the PyAnsys Geometry plotter.
----------------------	---

## Submodules

<code>plotter</code>	Provides plotting for various PyAnsys Geometry objects.
<code>plotter_helper</code>	Provides a wrapper to aid in plotting.
<code>plotting_types</code>	Data types for plotting.
<code>trame_gui</code>	Module for using <code>trame</code> for visualization.

## The widgets package

### Summary

### Submodules

<code>button</code>	Provides for implementing buttons in PyAnsys Geometry.
<code>displace_arrows</code>	Provides the displacement arrows widget for the PyVista plotter.
<code>measure</code>	Provides the ruler widget for the PyAnsys Geometry plotter.
<code>ruler</code>	Provides the ruler widget for the PyAnsys Geometry plotter.
<code>show_design_point</code>	Provides the ruler widget for the PyAnsys Geometry plotter.
<code>view_button</code>	Provides the view button widget for changing the camera view.
<code>widget</code>	Provides the abstract implementation of plotter widgets.

## The `button.py` module

### Summary

### Classes

<code>Button</code>	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.

#### Parameters

`plotter`  
[Plotter] Plotter to draw the buttons on.

`button_config`  
[tuple] Tuple containing the position and the path to the icon of the button.

## Notes

This class wraps the PyVista `add_checkbox_button_widget()` method.

## Overview

### Abstract methods

<code>callback</code>	Get the functionality of the button, which is implemented by subclasses.
-----------------------	--

### Methods

<code>update</code>	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

<code>DisplacementArrow</code>	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.

### Parameters

**plotter**  
[`Plotter`] Plotter to draw the buttons on.

**direction**  
[`CameraPanDirection`] Direction that the camera is to move.

## 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

<code>MeasureWidget</code>	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.

### Parameters

#### `plotter_helper`

[`PlotterHelper`] Provides the plotter to add the measure widget to.

### Overview

### Methods

<code>callback</code>	Remove or add the measurement widget actor upon click.
<code>update</code>	Define the measurement widget button params.

### Import detail

<code>from ansys.geometry.core.plotting.widgets.measure import MeasureWidget</code>
---

### 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

<code>Ruler</code>	Provides the ruler widget for the PyAnsys Geometry Plotter class.
--------------------	---

#### Ruler

`class Ruler(plotter: pv.Plotter)`

Bases: `ansys.geometry.core.plotting.widgets.widget.PlotterWidget`

Provides the ruler widget for the PyAnsys Geometry Plotter class.

##### Parameters

###### `plotter`

[`Plotter`] Provides the plotter to add the ruler widget to.

#### Overview

#### Methods

<code>callback</code>	Remove or add the ruler widget actor upon click.
-----------------------	--

<code>update</code>	Define the configuration and representation of the ruler widget button.
---------------------	---

#### Import detail

<code>from ansys.geometry.core.plotting.widgets.ruler import Ruler</code>
---

#### 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

<code>ShowDesignPoints</code>	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.

##### Parameters

`plotter_helper`

[`PlotterHelper`] Provides the plotter to add the button to.

#### Overview

#### Methods

<code>callback</code>	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

<code>ViewButton</code>	Provides for changing the view.
-------------------------	---------------------------------

### Enums

<code>ViewDirection</code>	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.

### Parameters

#### plotter

[Plotter] Plotter to draw the buttons on.

**direction**

[*ViewDirection*] Direction of the view.

## Overview

### Methods

<b>callback</b>	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

---

<code>XYPLUS</code>
<code>XYMINUS</code>
<code>XZPLUS</code>
<code>XZMINUS</code>
<code>YZPLUS</code>
<code>YZMINUS</code>
<code>ISOMETRIC</code>

---

## 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

<code>PlotterWidget</code>	Provides an abstract class for plotter widgets.
----------------------------	---

## PlotterWidget

```
class PlotterWidget(plotter: pyvista.Plotter)
```

Bases: `abc.ABC`

Provides an abstract class for plotter widgets.

### Parameters

`plotter`  
[`Plotter`] Plotter instance to add the widget to.

## Notes

These widgets are intended to be used with PyVista plotter objects. More specifically, the way in which this abstraction has been built ensures that these widgets are easily integrable with PyAnsys Geometry's own `Plotter` class.

## Overview

### Abstract methods

<code>callback</code>	General callback function for <code>PlotterWidget</code> objects.
<code>update</code>	General update function for <code>PlotterWidget</code> objects.

### Properties

<code>plotter</code>	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

<i>Plotter</i>	Provides for plotting sketches and bodies.
----------------	--

#### Constants

<code>DEFAULT_COLOR</code>	Default color we use for the plotter actors.
<code>PICKED_COLOR</code>	Color to use for the actors that are currently picked.
<code>EDGE_COLOR</code>	Default color to use for the edges.
<code>PICKED_EDGE_COLOR</code>	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.

##### Parameters

<b>scene</b>	[ <code>Plotter</code> , default: <code>None</code> ] Scene instance for rendering the objects.
<b>color_opts</b>	[ <code>dict</code> , default: <code>None</code> ] Dictionary containing the background and top colors.
<b>num_points</b>	[ <code>int</code> , default: 100] Number of points to use to render the shapes.
<b>enable_widgets: bool, default: True</b>	Whether to enable widget buttons in the plotter window. Widget buttons must be disabled when using <code>trame</code> for visualization.

## Overview

### Methods

<code>view_xy</code>	View the scene from the XY plane.
<code>view_xz</code>	View the scene from the XZ plane.
<code>view_yx</code>	View the scene from the YX plane.
<code>view_yz</code>	View the scene from the YZ plane.
<code>view_zx</code>	View the scene from the ZX plane.
<code>view_zy</code>	View the scene from the ZY plane.
<code>plot_frame</code>	Plot a frame in the scene.
<code>plot_plane</code>	Plot a plane in the scene.
<code>plot_sketch</code>	Plot a sketch in the scene.
<code>add_body_edges</code>	Add the outer edges of a body to the plot.
<code>add_body</code>	Add a body to the scene.
<code>add_component</code>	Add a component to the scene.
<code>add_sketch_polydata</code>	Add sketches to the scene from PyVista polydata.
<code>clip</code>	Clip the passed mesh with a plane.
<code>add_design_point</code>	Add a DesignPoint object to the plotter.
<code>add</code>	Add any type of object to the scene.
<code>add_list</code>	Add a list of any type of object to the scene.
<code>show</code>	Show the rendered scene on the screen.

### Properties

<code>scene</code>	Rendered scene object.
<code>geom_object_actors_map</code>	Mapping between the ~pyvista.Actor and the PyAnsys Geometry objects.

### 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.

`property Plotter.geom_object_actors_map: beartype.typing.Dict[pyvista.Actor, ansys.geometry.core.plotting.plotting_types.GeoObjectPlot]`

Mapping between the ~pyvista.Actor and the PyAnsys Geometry objects.

## 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.GemObjectPlot, **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)`  
 $\rightarrow \text{None}$

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.clip(mesh: beartype.typing.Union[pyvista.PolyData, pyvista.MultiBlock], plane:  
 ansys.geometry.core.math.plane.Plane = None) \rightarrow beartype.typing.Union[pyvista.PolyData,  
 pyvista.MultiBlock]`

Clip the passed mesh with a plane.

**Parameters****mesh**

`[Union[pv.PolyData, pv.MultiBlock]]` Mesh you want to clip.

**normal**

`[str, optional]` Plane you want to use for clipping, by default “x”. Available options: [“x”, “-x”, “y”, “-y”, “z”, “-z”]

**origin**

`[tuple, optional]` Origin point of the plane, by default None

**Returns****Union[pv.PolyData,pv.MultiBlock]**

The clipped mesh.

`Plotter.add_design_point(design_point: ansys.geometry.core.designer.designpoint.DesignPoint,  
 **plotting_options) \rightarrow \text{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) \rightarrow \text{None}`

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.

`Plotter.add_list(plotting_list: beartype.typing.List[beartype.typing.Any], merge_bodies: bool = False, merge_components: bool = False, filter: str = None, **plotting_options) → None`

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.

`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

---

<code>PlotterHelper</code>	Provides for simplifying the selection of trame in <code>plot()</code> 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.

### Parameters

#### `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.

#### `allow_picking: bool, default: False`

Enables/disables the picking capabilities in the PyVista plotter.

## Overview

### Methods

<code>enable_widgets</code>	Enable the widgets for the plotter.
<code>select_object</code>	Select an object in the plotter.
<code>unselect_object</code>	Unselect an object in the plotter.
<code>picker_callback</code>	Define callback for the element picker.
<code>compute_edge_object_map</code>	Compute the mapping between plotter actors and EdgePlot objects.
<code>enable_picking</code>	Enable picking capabilities in the plotter.
<code>disable_picking</code>	Disable picking capabilities in the plotter.
<code>add</code>	Add a pyansys-geometry or PyVista object to the plotter.
<code>plot</code>	Plot and show any PyAnsys Geometry object.
<code>show_plotter</code>	Show the plotter or start the <code>trame</code> 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*:

*bertype.typing.Union[ansys.geometry.core.plotting.plotting\_types.G 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*:

*bertype.typing.Union[ansys.geometry.core.plotting.plotting\_types.G 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.add(object: beartype.typing.Any, **plotting_options)`

Add a pyansys-geometry or PyVista object to the plotter.

#### Parameters

##### `object`

[`Any`] Object you want to show.

`PlotterHelper.plot(object: beartype.typing.Any = None, 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`, default: `None`] 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**

<code>EdgePlot</code>	Mapper class to relate PyAnsys Geometry edges with its PyVista actor.
<code>GeomObjectPlot</code>	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.

**Parameters****actor**

[`Actor`] PyVista actor that represents the edge.

**edge\_object**

[`Edge`] PyAnsys Geometry edge that is represented by the PyVista actor.

**parent**

[`GeomObjectPlot`, optional] Parent PyAnsys Geometry object of this edge, by default `None`.

## Overview

### Properties

<code>actor</code>	Return PyVista actor of the object.
<code>edge_object</code>	Return the PyAnsys Geometry edge.
<code>parent</code>	Parent PyAnsys Geometry object of this edge.
<code>name</code>	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.

### Parameters

#### actor

[pv.Actor] PyVista actor that represents the pyansys-geometry object.

#### object

[Any] PyAnsys Geometry object that is represented.

#### edges

[List[EdgePlot], optional] List of edges of the PyAnsys Geometry object, by default None.

#### add\_body\_edges: bool, optional

Flag to specify if you want to be able to add edges.

## 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

<code>TrameVisualizer</code>	Defines the trame layout view.
------------------------------	--------------------------------

### TrameVisualizer

**class** `TrameVisualizer`

Defines the trame layout view.

### Overview

### Methods

<code>set_scene</code>	Set the trame layout view and the mesh to show through the PyVista plotter.
<code>show</code>	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

---

<code>circle</code>	Provides for creating and managing a circle.
<code>cone</code>	Provides for creating and managing a cone.
<code>curve_evaluation</code>	Provides for creating and managing a curve.
<code>cylinder</code>	Provides for creating and managing a cylinder.
<code>ellipse</code>	Provides for creating and managing an ellipse.
<code>line</code>	Provides for creating and managing a line.
<code>parameterization</code>	Provides the parametrization-related classes.
<code>sphere</code>	Provides for creating and managing a sphere.
<code>surface_evaluation</code>	Provides for evaluating a surface.
<code>torus</code>	Provides for creating and managing a torus.

---

## The circle.py module

### Summary

### Classes

<code>Circle</code>	Provides 3D circle representation.
<code>CircleEvaluation</code>	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.

#### Parameters

##### origin

[Union[ndarray, RealSequence, Point3D]] Origin of the circle.

##### radius

[Union[Quantity, Distance, Real]] Radius of the circle.

##### reference

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] X-axis direction.

##### axis

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] Z-axis direction.

### Overview

### Methods

<code>evaluate</code>	Evaluate the circle at a given parameter.
<code>transformed_copy</code>	Create a transformed copy of the circle based on a transformation matrix.
<code>mirrored_copy</code>	Create a mirrored copy of the circle along the y-axis.
<code>project_point</code>	Project a point onto the circle and evaluate the circle.
<code>is_coincident_circle</code>	Determine if the circle is coincident with another.
<code>get_parameterization</code>	Get the parametrization of the circle.

## Properties

<code>origin</code>	Origin of the circle.
<code>radius</code>	Radius of the circle.
<code>diameter</code>	Diameter of the circle.
<code>perimeter</code>	Perimeter of the circle.
<code>area</code>	Area of the circle.
<code>dir_x</code>	X-direction of the circle.
<code>dir_y</code>	Y-direction of the circle.
<code>dir_z</code>	Z-direction of the circle.

## Special methods

<code>__eq__</code>	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 evaluate 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 parameterization 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.

#### **Parameters**

##### **circle: ~ansys.geometry.core.primitives.circle.Circle**

Circle to evaluate.

##### **parameter: Real**

Parameter to evaluate the circle at.

## **Overview**

## **Methods**

<code>position</code>	Position of the evaluation.
<code>tangent</code>	Tangent of the evaluation.
<code>normal</code>	Normal to the circle.
<code>first_derivative</code>	First derivative of the evaluation.
<code>second_derivative</code>	Second derivative of the evaluation.
<code>curvature</code>	Curvature of the circle.

## **Properties**

<code>circle</code>	Circle being evaluated.
<code>parameter</code>	Parameter that the evaluation is based upon.

## Import detail

```
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

<code>Cone</code>	Provides 3D cone representation.
<code>ConeEvaluation</code>	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.

#### Parameters

##### `origin`

[Union[ndarray, RealSequence, Point3D]] Origin of the cone.

##### `radius`

[Union[Quantity, Distance, Real]] Radius of the cone.

##### `half_angle`

[Union[Quantity, Angle, Real]] Half angle of the apex, determining the upward angle.

##### `reference`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] X-axis direction.

##### `axis`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] Z-axis direction.

## Overview

### Methods

<code>transformed_copy</code>	Create a transformed copy of the cone based on a transformation matrix.
<code>mirrored_copy</code>	Create a mirrored copy of the cone along the y-axis.
<code>evaluate</code>	Evaluate the cone at given parameters.
<code>project_point</code>	Project a point onto the cone and evaluate the cone.
<code>get_u_parameterization</code>	Get the parametrization conditions for the U parameter.
<code>get_v_parameterization</code>	Get the parametrization conditions for the V parameter.

### Properties

<code>origin</code>	Origin of the cone.
<code>radius</code>	Radius of the cone.
<code>half_angle</code>	Half angle of the apex.
<code>dir_x</code>	X-direction of the cone.
<code>dir_y</code>	Y-direction of the cone.
<code>dir_z</code>	Z-direction of the cone.
<code>height</code>	Height of the cone.
<code>surface_area</code>	Surface area of the cone.
<code>volume</code>	Volume of the cone.
<code>apex</code>	Apex point of the cone.
<code>apex_param</code>	Apex parameter of the cone.

### Special methods

<code>__eq__</code>	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.

**Parameters****cone: ~ansys.geometry.core.primitives.cone.Cone**

Cone to evaluate.

**parameter: ParamUV**

Parameters (u, v) to evaluate the cone at.

## Overview

### Methods

<i>position</i>	Position of the evaluation.
<i>normal</i>	Normal to the surface.
<i>u_derivative</i>	First derivative with respect to the U parameter.
<i>v_derivative</i>	First derivative with respect to the V parameter.
<i>uu_derivative</i>	Second derivative with respect to the U parameter.
<i>uv_derivative</i>	Second derivative with respect to the U and V parameters.
<i>vv_derivative</i>	Second derivative with respect to the V parameter.
<i>min_curvature</i>	Minimum curvature of the cone.
<i>min_curvature_direction</i>	Minimum curvature direction.
<i>max_curvature</i>	Maximum curvature of the cone.
<i>max_curvature_direction</i>	Maximum curvature direction.

### Properties

<i>cone</i>	Cone being evaluated.
<i>parameter</i>	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

<code>CurveEvaluation</code>	Provides for evaluating a curve.
------------------------------	----------------------------------

### CurveEvaluation

`class CurveEvaluation(parameter: ansys.geometry.core.typing.Real = None)`

Provides for evaluating a curve.

### Overview

### Abstract methods

<code>position</code>	Position of the evaluation.
<code>first_derivative</code>	First derivative of the evaluation.
<code>second_derivative</code>	Second derivative of the evaluation.
<code>curvature</code>	Curvature of the evaluation.

### Methods

<code>is_set</code>	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.

### Parameters

<b>origin</b>	[Union[ <a href="#">ndarray</a> , <a href="#">RealSequence</a> , <a href="#">Point3D</a> ]] Origin of the cylinder.
<b>radius</b>	[Union[ <a href="#">Quantity</a> , <a href="#">Distance</a> , <a href="#">Real</a> ]] Radius of the cylinder.
<b>reference</b>	[Union[ <a href="#">ndarray</a> , <a href="#">RealSequence</a> , <a href="#">UnitVector3D</a> , <a href="#">Vector3D</a> ]] X-axis direction.
<b>axis</b>	[Union[ <a href="#">ndarray</a> , <a href="#">RealSequence</a> , <a href="#">UnitVector3D</a> , <a href="#">Vector3D</a> ]] Z-axis direction.

## Overview

### Methods

<code>surface_area</code>	Get the surface area of the cylinder.
<code>volume</code>	Get the volume of the cylinder.
<code>transformed_copy</code>	Create a transformed copy of the cylinder based on a transformation matrix.
<code>mirrored_copy</code>	Create a mirrored copy of the cylinder along the y-axis.
<code>evaluate</code>	Evaluate the cylinder at the given parameters.
<code>project_point</code>	Project a point onto the cylinder and evaluate the cylinder.
<code>get_u_parameterization</code>	Get the parametrization conditions for the U parameter.
<code>get_v_parameterization</code>	Get the parametrization conditions for the V parameter.

### Properties

<code>origin</code>	Origin of the cylinder.
<code>radius</code>	Radius of the cylinder.
<code>dir_x</code>	X-direction of the cylinder.
<code>dir_y</code>	Y-direction of the cylinder.
<code>dir_z</code>	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]*)  
 $\rightarrow$  *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.

**Parameters****cylinder: Cylinder**

Cylinder to evaluate.

**parameter: ParamUV**

Parameters (u, v) to evaluate the cylinder at.

## Overview

### Methods

<i>position</i>	Position of the evaluation.
<i>normal</i>	Normal to the surface.
<i>u_derivative</i>	First derivative with respect to the U parameter.
<i>v_derivative</i>	First derivative with respect to the V parameter.
<i>uu_derivative</i>	Second derivative with respect to the U parameter.
<i>uv_derivative</i>	Second derivative with respect to the U and V parameters.
<i>vv_derivative</i>	Second derivative with respect to the V parameter.
<i>min_curvature</i>	Minimum curvature of the cylinder.
<i>min_curvature_direction</i>	Minimum curvature direction.
<i>max_curvature</i>	Maximum curvature of the cylinder.
<i>max_curvature_direction</i>	Maximum curvature direction.

### Properties

<i>cylinder</i>	Cylinder being evaluated.
<i>parameter</i>	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**

<code>Ellipse</code>	Provides 3D ellipse representation.
<code>EllipseEvaluation</code>	Evaluate an ellipse at a given parameter.

**Ellipse**

```
class Ellipse(origin: beartype.typing.Union[numumpy.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[numumpy.ndarray, ansys.geometry.core.typing.RealSequence,
    ansys.geometry.core.math.vector.UnitVector3D, ansys.geometry.core.math.vector.Vector3D] =
    UNITVECTOR3D_X, axis: beartype.typing.Union[numumpy.ndarray,
    ansys.geometry.core.typing.RealSequence, ansys.geometry.core.math.vector.UnitVector3D,
    ansys.geometry.core.math.vector.Vector3D] = UNITVECTOR3D_Z)
```

Provides 3D ellipse representation.

**Parameters****origin**

[Union[ndarray, RealSequence, Point3D]] Origin of the ellipse.

**major\_radius**

[Union[Quantity, Distance, Real]] Major radius of the ellipse.

**minor\_radius**

[Union[Quantity, Distance, Real]] Minor radius of the ellipse.

**reference**

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] X-axis direction.

**axis**

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] Z-axis direction.

**Overview****Methods**

<code>mirrored_copy</code>	Create a mirrored copy of the ellipse along the y-axis.
<code>evaluate</code>	Evaluate the ellipse at the given parameter.
<code>project_point</code>	Project a point onto the ellipse and evaluate the ellipse.
<code>is_coincident_ellipse</code>	Determine if this ellipse is coincident with another.
<code>transformed_copy</code>	Create a transformed copy of the ellipse based on a transformation matrix.
<code>get_parameterization</code>	Get the parametrization of the ellipse.

**Properties**

<code>origin</code>	Origin of the ellipse.
<code>major_radius</code>	Major radius of the ellipse.
<code>minor_radius</code>	Minor radius of the ellipse.
<code>dir_x</code>	X-direction of the ellipse.
<code>dir_y</code>	Y-direction of the ellipse.
<code>dir_z</code>	Z-direction of the ellipse.
<code>eccentricity</code>	Eccentricity of the ellipse.
<code>linear_eccentricity</code>	Linear eccentricity of the ellipse.
<code>semi_latus_rectum</code>	Semi-latus rectum of the ellipse.
<code>perimeter</code>	Perimeter of the ellipse.
<code>area</code>	Area of the ellipse.

**Special methods**

<code>__eq__</code>	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.

### Parameters

**ellipse: Ellipse**

Ellipse to evaluate.

**parameter: float, int**

Parameter to evaluate the ellipse at.

## Overview

### Methods

<code>position</code>	Position of the evaluation.
<code>tangent</code>	Tangent of the evaluation.
<code>normal</code>	Normal of the evaluation.
<code>first_derivative</code>	Girst derivative of the evaluation.
<code>second_derivative</code>	Second derivative of the evaluation.
<code>curvature</code>	Curvature of the ellipse.

### Properties

<code>ellipse</code>	Ellipse being evaluated.
<code>parameter</code>	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`

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.

`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

<code>Line</code>	Provides 3D line representation.
<code>LineEvaluation</code>	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.

#### Parameters

##### `origin`

[Union[ndarray, RealSequence, Point3D]] Origin of the line.

##### `direction`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] Direction of the line.

### Overview

### Methods

<code>evaluate</code>	Evaluate the line at a given parameter.
<code>transformed_copy</code>	Create a transformed copy of the line based on a transformation matrix.
<code>project_point</code>	Project a point onto the line and evaluate the line.
<code>is_coincident_line</code>	Determine if the line is coincident with another line.
<code>is_opposite_line</code>	Determine if the line is opposite another line.
<code>get_parameterization</code>	Get the parametrization of the line.

## Properties

<code>origin</code>	Origin of the line.
<code>direction</code>	Direction of the line.

## Special methods

<code>__eq__</code>	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 *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

<code>position</code>	Position of the evaluation.
<code>tangent</code>	Tangent of the evaluation, which is always equal to the direction of the line.
<code>first_derivative</code>	First derivative of the evaluation.
<code>second_derivative</code>	Second derivative of the evaluation.
<code>curvature</code>	Curvature of the line, which is always 0.

### Properties

<code>line</code>	Line being evaluated.
<code>parameter</code>	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

---

<code>ParamUV</code>	Parameter class containing 2 parameters: (u, v).
<code>Interval</code>	Interval class that defines a range of values.
<code>Parameterization</code>	Parameterization class describes the parameters of a specific geometry.

---

### Enums

---

<code>ParamForm</code>	ParamForm enum class that defines the form of a Parameterization.
<code>ParamType</code>	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*).

### Parameters

**u**

[Real] u-parameter.

**v**

[Real] v-parameter.

### Notes

Likened to a 2D point in UV space Used as an argument in parametric surface evaluations. This matches the service implementation for the Geometry service.

## Overview

### Properties

<b>u</b>	u-parameter.
<b>v</b>	v-parameter.

### Special methods

<b>__add__</b>	Add the <i>u</i> and <i>v</i> components of the other ParamUV to this ParamUV.
<b>__sub__</b>	Subtract the <i>u</i> and <i>v</i> components of the other ParamUV from this ParamUV.
<b>__mul__</b>	Multiplies the <i>u</i> and <i>v</i> components of this ParamUV by the other ParamUV.
<b>__truediv__</b>	Divides the <i>u</i> and <i>v</i> components of this ParamUV by the other ParamUV.
<b>__repr__</b>	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.

### Parameters

#### start

[Real] Start value of the interval.

#### end

[Real] End value of the interval.

## Overview

## Methods

<i>is_open</i>	If the interval is open (-inf, inf).
----------------	--------------------------------------

<i>is_closed</i>	If the interval is closed. Neither value is inf or -inf.
------------------	--

<i>get_span</i>	Return the quantity contained by the interval. Interval must be closed.
-----------------	---

## Properties

<i>start</i>	Start value of the interval.
--------------	------------------------------

<i>end</i>	End value of the interval.
------------	----------------------------

## Special methods

<i>__repr__</i>	Represent the Interval as a string.
-----------------	-------------------------------------

## Import detail

<code>from ansys.geometry.core.primitives.parameterization import Interval</code>
---

## 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.

### Parameters

`form`

[`ParamForm`] Form of the parameterization.

`type`

[`ParamType`] Type of the parameterization.

`interval`

[`Interval`] Interval of the parameterization.

## Overview

## Properties

<code>form</code>	The form of the parameterization.
<code>type</code>	The type of the parameterization.
<code>interval</code>	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

<i>LINEAR</i>
<i>CIRCULAR</i>
<i>OTHER</i>

## 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

<code>Sphere</code>	Provides 3D sphere representation.
<code>SphereEvaluation</code>	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.

##### Parameters

###### `origin`

[Union[ndarray, RealSequence, Point3D]] Origin of the sphere.

###### `radius`

[Union[Quantity, Distance, Real]] Radius of the sphere.

###### `reference`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] X-axis direction.

###### `axis`

[Union[ndarray, RealSequence, UnitVector3D, Vector3D]] Z-axis direction.

#### Overview

#### Methods

<code>transformed_copy</code>	Create a transformed copy of the sphere based on a transformation matrix.
<code>mirrored_copy</code>	Create a mirrored copy of the sphere along the y-axis.
<code>evaluate</code>	Evaluate the sphere at the given parameters.
<code>project_point</code>	Project a point onto the sphere and evaluate the sphere.
<code>get_u_parameterization</code>	Get the parametrization conditions for the U parameter.
<code>get_v_parameterization</code>	Get the parametrization conditions for the V parameter.

## Properties

<code>origin</code>	Origin of the sphere.
<code>radius</code>	Radius of the sphere.
<code>dir_x</code>	X-direction of the sphere.
<code>dir_y</code>	Y-direction of the sphere.
<code>dir_z</code>	Z-direction of the sphere.
<code>surface_area</code>	Surface area of the sphere.
<code>volume</code>	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 parameterization 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 parameterization 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.

#### Parameters

**sphere:** `~ansys.geometry.core.primitives.sphere.Sphere`  
Sphere to evaluate.

**parameter:** `ParamUV`  
Parameters (u, v) to evaluate the sphere at.

## Overview

## Methods

<code>position</code>	Position of the evaluation.
<code>normal</code>	The normal to the surface.
<code>u_derivative</code>	First derivative with respect to the U parameter.
<code>v_derivative</code>	First derivative with respect to the V parameter.
<code>uu_derivative</code>	Second derivative with respect to the U parameter.
<code>uv_derivative</code>	Second derivative with respect to the U and V parameters.
<code>vv_derivative</code>	Second derivative with respect to the V parameter.
<code>min_curvature</code>	Minimum curvature of the sphere.
<code>min_curvature_direction</code>	Minimum curvature direction.
<code>max_curvature</code>	Maximum curvature of the sphere.
<code>max_curvature_direction</code>	Maximum curvature direction.

## Properties

<code>sphere</code>	Sphere being evaluated.
<code>parameter</code>	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

<code>SurfaceEvaluation</code>	Provides for evaluating a surface.
--------------------------------	------------------------------------

### SurfaceEvaluation

`class SurfaceEvaluation(parameter: ansys.geometry.core.primitives.parameterization.ParamUV)`

Provides for evaluating a surface.

### Overview

#### Abstract methods

<code>position</code>	Point on the surface, based on the evaluation.
<code>normal</code>	Normal to the surface.
<code>u_derivative</code>	First derivative with respect to the U parameter.
<code>v_derivative</code>	First derivative with respect to the V parameter.
<code>uu_derivative</code>	Second derivative with respect to the U parameter.
<code>uv_derivative</code>	The second derivative with respect to the U and V parameters.
<code>vv_derivative</code>	The second derivative with respect to v.
<code>min_curvature</code>	Minimum curvature.
<code>min_curvature_direction</code>	Minimum curvature direction.
<code>max_curvature</code>	Maximum curvature.
<code>max_curvature_direction</code>	Maximum curvature direction.

### Properties

<code>parameter</code>	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

<code>Torus</code>	Provides 3D torus representation.
<code>TorusEvaluation</code>	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.

#### Parameters

**origin**  
`[Union[ndarray, RealSequence, Point3D]]` Centered origin of the torus.

**direction\_x**  
`[Union[ndarray, RealSequence, UnitVector3D, Vector3D]]` X-axis direction.

**direction\_y**  
`[Union[ndarray, RealSequence, UnitVector3D, Vector3D]]` Y-axis direction.

**major\_radius**  
`[Union[Quantity, Distance, Real]]` Major radius of the torus.

**minor\_radius**  
`[Union[Quantity, Distance, Real]]` Minor radius of the torus.

## Overview

### Methods

<code>transformed_copy</code>	Create a transformed copy of the torus based on a transformation matrix.
<code>mirrored_copy</code>	Create a mirrored copy of the torus along the y-axis.
<code>evaluate</code>	Evaluate the torus at the given parameters.
<code>get_u_parameterization</code>	Get the parametrization conditions for the U parameter.
<code>get_v_parameterization</code>	Get the parametrization conditions of the V parameter.
<code>project_point</code>	Project a point onto the torus and evaluate the torus.

### Properties

<code>origin</code>	Origin of the torus.
<code>major_radius</code>	Semi-major radius of the torus.
<code>minor_radius</code>	Semi-minor radius of the torus.
<code>dir_x</code>	X-direction of the torus.
<code>dir_y</code>	Y-direction of the torus.
<code>dir_z</code>	Z-direction of the torus.
<code>volume</code>	Volume of the torus.
<code>surface_area</code>	Surface_area of the torus.

### Special methods

<code>__eq__</code>	Equals operator for the Torus class.
---------------------	--------------------------------------

### Import detail

```
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 parameterization 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.

**Parameters**

**Torus: ~ansys.geometry.core.primitives.torus.Torus**

Torust to evaluate.

**parameter: ParamUV**

Parameters (u, v) to evaluate the torus at.

**Overview**

## Methods

<code>position</code>	Position of the evaluation.
<code>normal</code>	Normal to the surface.
<code>u_derivative</code>	First derivative with respect to the U parameter.
<code>v_derivative</code>	First derivative with respect to the V parameter.
<code>uu_derivative</code>	Second derivative with respect to the U parameter.
<code>uv_derivative</code>	Second derivative with respect to the U and V parameters.
<code>vv_derivative</code>	Second derivative with respect to the V parameter.
<code>curvature</code>	Curvature of the torus.
<code>min_curvature</code>	Minimum curvature of the torus.
<code>min_curvature_direction</code>	Minimum curvature direction.
<code>max_curvature</code>	Maximum curvature of the torus.
<code>max_curvature_direction</code>	Maximum curvature direction.

## Properties

<code>torus</code>	Torus being evaluated.
<code>parameter</code>	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**

<b><i>arc</i></b>	Provides for creating and managing an arc.
<b><i>box</i></b>	Provides for creating and managing a box (quadrilateral).
<b><i>circle</i></b>	Provides for creating and managing a circle.
<b><i>edge</i></b>	Provides for creating and managing an edge.
<b><i>ellipse</i></b>	Provides for creating and managing an ellipse.
<b><i>face</i></b>	Provides for creating and managing a face (closed 2D sketch).
<b><i>gears</i></b>	Module for creating and managing gears.
<b><i>polygon</i></b>	Provides for creating and managing a polygon.
<b><i>segment</i></b>	Provides for creating and managing a segment.
<b><i>sketch</i></b>	Provides for creating and managing a sketch.
<b><i>slot</i></b>	Provides for creating and managing a slot.
<b><i>trapezoid</i></b>	Provides for creating and managing a trapezoid.
<b><i>triangle</i></b>	Provides for creating and managing a triangle.

## The `arc.py` module

### Summary

### Classes

<code>Arc</code>	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.

#### Parameters

##### `center`

[`Point2D`] Center point of the arc.

##### `start`

[`Point2D`] Starting point of the arc.

##### `end`

[`Point2D`] Ending point of the arc.

##### `clockwise`

[`bool`, default: `False`] Whether the arc spans the clockwise angle between the start and end points. When `False` (default), the arc spans the counter-clockwise angle. When `True`, the arc spans the clockwise angle.

### Overview

### Constructors

<code>from_three_points</code>	Create an arc from three given points.
--------------------------------	--

### Properties

<code>start</code>	Starting point of the arc line.
<code>end</code>	Ending point of the arc line.
<code>length</code>	Length of the arc.
<code>radius</code>	Radius of the arc.
<code>center</code>	Center point of the arc.
<code>angle</code>	Angle of the arc.
<code>is_clockwise</code>	Flag indicating whether the rotation of the angle is clockwise.
<code>sector_area</code>	Area of the sector of the arc.
<code>visualization_polydata</code>	VTK polydata representation for PyVista visualization.

## Special methods

<code>__eq__</code>	Equals operator for the Arc class.
<code>__ne__</code>	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

<code>Box</code>	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.

### 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, Angle, Real], default: 0] Placement angle for orientation alignment.

## Overview

## Properties

<code>center</code>	Center point of the box.
<code>width</code>	Width of the box.
<code>height</code>	Height of the box.
<code>perimeter</code>	Perimeter of the box.
<code>area</code>	Area of the box.
<code>visualization_polydata</code>	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: pvista.PolyData`

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global cartesian coordinate system.

### Returns

`pvista.PolyData`

VTK pvista.Polydata configuration.

## Description

Provides for creating and managing a box (quadrilateral).

## The `circle.py` module

### Summary

### Classes

<code>SketchCircle</code>	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.

### Parameters

**center: Point2D**

Center point of the circle.

**radius**

[Union[Quantity, Distance, Real]] Radius of the circle.

**plane**

[Plane, optional] Plane containing the sketched circle, which is the global XY plane by default.

**Overview****Methods**


---

<i>plane_change</i>	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 SketchFace.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

<code>SketchEdge</code>	Provides for modeling edges forming sketched shapes.
-------------------------	--

## SketchEdge

### `class SketchEdge`

Provides for modeling edges forming sketched shapes.

### Overview

### Methods

<code>plane_change</code>	Redefine the plane containing <code>SketchEdge</code> objects.
---------------------------	--

## Properties

<code>start</code>	Starting point of the edge.
<code>end</code>	Ending point of the edge.
<code>length</code>	Length of the edge.
<code>visualization_polydata</code>	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 method 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

<code>SketchEllipse</code>	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`

Provides for modeling an ellipse.

#### Parameters

##### `center: Point2D`

Center point of the ellipse.

##### `major_radius`

[Union[Quantity, Distance, Real]] Major radius of the ellipse.

##### `minor_radius`

[Union[Quantity, Distance, Real]] Minor radius of the ellipse.

##### `angle`

[Union[Quantity, Angle, Real], default: 0] Placement angle for orientation alignment.

##### `plane`

[Plane, optional] Plane containing the sketched ellipse, which is the global XY plane by default.

## Overview

## Methods

<code>plane_change</code>	Redefine the plane containing SketchEllipse objects.
---------------------------	--

## Properties

<code>center</code>	Center point of the ellipse.
<code>angle</code>	Orientation angle of the ellipse.
<code>perimeter</code>	Perimeter of the circle.
<code>visualization_polydata</code>	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 SketchFace.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) → 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

<code>edges</code>	List of all component edges forming the face.
<code>perimeter</code>	Perimeter of the face.
<code>visualization_polydata</code>	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

<code>Gear</code>	Provides the base class for sketching gears.
<code>DummyGear</code>	Provides the dummy class for sketching gears.
<code>SpurGear</code>	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

<code>visualization_polydata</code>	VTK polydata representation for PyVista visualization.
-------------------------------------	--

### Import detail

<code>from ansys.geometry.core.sketch.gears import Gear</code>
--

### 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.

### 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.

### 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.

### 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.

## Overview

### Properties

<code>origin</code>	Origin of the spur gear.
<code>module</code>	Module of the spur gear.
<code>pressure_angle</code>	Pressure angle of the spur gear.
<code>n_teeth</code>	Number of teeth of the spur gear.
<code>ref_diameter</code>	Reference diameter of the spur gear.
<code>base_diameter</code>	Base diameter of the spur gear.
<code>addendum</code>	Addendum of the spur gear.
<code>dedendum</code>	Dedendum of the spur gear.
<code>tip_diameter</code>	Tip diameter of the spur gear.
<code>root_diameter</code>	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

<code>Polygon</code>	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.

#### Parameters

##### `center: Point2D`

Center point of the circle.

##### `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.

## Overview

## Properties

<code>center</code>	Center point of the polygon.
<code>inner_radius</code>	Inner radius (apothem) of the polygon.
<code>n_sides</code>	Number of sides of the polygon.
<code>angle</code>	Orientation angle of the polygon.
<code>length</code>	Side length of the polygon.
<code>outer_radius</code>	Outer radius of the polygon.
<code>perimeter</code>	Perimeter of the polygon.
<code>area</code>	Area of the polygon.
<code>visualization_polydata</code>	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

<code>SketchSegment</code>	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.

##### Parameters

###### `start`

[`Point2D`] Starting point of the line segment.

###### `end`

[`Point2D`] Ending point of the line segment.

###### `plane`

[`Plane`, optional] Plane containing the sketched circle, which is the global XY plane by default.

#### Overview

#### Methods

<code>plane_change</code>	Redefine the plane containing <code>SketchSegment</code> objects.
---------------------------	---

## Properties

<code>start</code>	Starting point of the segment.
<code>end</code>	Ending point of the segment.
<code>length</code>	Length of the segment.
<code>visualization_polydata</code>	VTK polydata representation for PyVista visualization.

## Special methods

<code>__eq__</code>	Equals operator for the SketchSegment class.
<code>__ne__</code>	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**

<i>Sketch</i>	Provides for building 2D sketch elements.
---------------	---

**Attributes**

<i>SketchObject</i>	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

<code>translate_sketch_plane</code>	Translate the origin location of the active sketch plane.
<code>translate_sketch_plane_by_offset</code>	Translate the origin location of the active sketch plane by offsets.
<code>translate_sketch_plane_by_distance</code>	Translate the origin location active sketch plane by distance.
<code>get</code>	Get a list of shapes with a given tag.
<code>face</code>	Add a sketch face to the sketch.
<code>edge</code>	Add a sketch edge to the sketch.
<code>select</code>	Add all objects that match provided tags to the current context.
<code>segment</code>	Add a segment sketch object to the sketch plane.
<code>segment_to_point</code>	Add a segment to the sketch plane starting from the previous edge end point.
<code>segment_from_point_and_vector</code>	Add a segment to the sketch starting from a given starting point.
<code>segment_from_vector</code>	Add a segment to the sketch starting from the end point of the previous edge.
<code>arc</code>	Add an arc to the sketch plane.
<code>arc_to_point</code>	Add an arc to the sketch starting from the end point of the previous edge.
<code>arc_from_three_points</code>	Add an arc to the sketch plane from three given points.
<code>triangle</code>	Add a triangle to the sketch using given vertex points.
<code>trapezoid</code>	Add a trapezoid to the sketch using given vertex points.
<code>circle</code>	Add a circle to the plane at a given center.
<code>box</code>	Create a box on the sketch.
<code>slot</code>	Create a slot on the sketch.
<code>ellipse</code>	Create an ellipse on the sketch.
<code>polygon</code>	Create a polygon on the sketch.
<code>dummy_gear</code>	Create a dummy gear on the sketch.
<code>spur_gear</code>	Create a spur gear on the sketch.
<code>tag</code>	Add a tag to the active selection of sketch objects.
<code>plot</code>	Plot all objects of the sketch to the scene.
<code>plot_selection</code>	Plot the current selection to the scene.
<code>sketch_polydata</code>	Get polydata configuration for all objects of the sketch to the scene.
<code>sketch_polydata_faces</code>	Get polydata configuration for all faces of the sketch to the scene.
<code>sketch_polydata_edges</code>	Get polydata configuration for all edges of the sketch to the scene.

## Properties

<code>plane</code>	Sketch plane configuration.
<code>edges</code>	List of all independently sketched edges.
<code>faces</code>	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 trapezoid 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[pv.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

<b>Slot</b>	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.

#### Parameters

**center**: :class:`Point2D <ansys.geometry.core.math.point.Point2D>`

Center point of the slot.

**width**

[Union[Quantity, Distance, Real]] Width of the slot main body.

**height**

[Union[Quantity, Distance, Real]] Height of the slot.

**angle**

[Union[Quantity, Angle, Real], default: 0] Placement angle for orientation alignment.

### Overview

### Properties

<b>center</b>	Center of the slot.
<b>width</b>	Width of the slot.
<b>height</b>	Height of the slot.
<b>perimeter</b>	Perimeter of the slot.
<b>area</b>	Area of the slot.
<b>visualization_polydata</b>	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: pvista.PolyData**

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

### Returns

**pvista.PolyData**

VTK pvista.Polydata configuration.

## Description

Provides for creating and managing a slot.

## The trapezoid.py module

### Summary

### Classes

<i>Trapezoid</i>	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.

### Parameters

#### width

[Union[Quantity, Distance, Real]] Width of the trapezoid.

#### height

[Union[Quantity, Distance, Real]] Height of the trapezoid.

#### 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: ZERO\_POINT2D

Center point of the trapezoid.

#### angle

[Union[Quantity, Angle, Real], default: 0] Placement angle for orientation alignment.

### Notes

If a nonsymmetrical slant angle is defined, the slant angle is applied to the left-most angle, and the non-symmetrical slant angle is applied to the right-most angle.

## Overview

## Properties

<code>center</code>	Center of the trapezoid.
<code>width</code>	Width of the trapezoid.
<code>height</code>	Height of the trapezoid.
<code>visualization_polydata</code>	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: pvista.PolyData`

VTK polydata representation for PyVista visualization.

The representation lies in the X/Y plane within the standard global Cartesian coordinate system.

### Returns

`pvista.PolyData`

VTK pvista.Polydata configuration.

## Description

Provides for creating and managing a trapezoid.

## The triangle.py module

### Summary

### Classes

<code>Triangle</code>	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.

### Parameters

**point1: Point2D**

Point that represents a triangle vertex.

**point2: Point2D**

Point that represents a triangle vertex.

**point3: Point2D**

Point that represents a triangle vertex.

## Overview

## Properties

<i>point1</i>	Triangle vertex 1.
<i>point2</i>	Triangle vertex 2.
<i>point3</i>	Triangle vertex 3.
<i>visualization_polydata</i>	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

<code>measurement_tools</code>	Provides tools for measurement.
<code>problem_areas</code>	The problem area definition.
<code>repair_tool_message</code>	Module for repair tool message.
<code>repair_tools</code>	Provides tools for repairing bodies.

### The `measurement_tools.py` module

#### Summary

#### Classes

<code>Gap</code>	Represents a gap between two bodies.
<code>MeasurementTools</code>	Measurement tools for PyAnsys Geometry.

#### Gap

`class Gap(distance: ansys.geometry.core.misc.measurements.Distance)`

Represents a gap between two bodies.

##### Parameters

###### `distance`

[*Distance*] Distance between two sides of the gap.

#### Overview

#### Properties

<code>distance</code>	Returns the closest distance between two bodies.
-----------------------	--

## Import detail

```
from ansys.geometry.core.tools.measurement_tools import Gap
```

## Property detail

### property `Gap.distance: Distance`

Returns the closest distance between two bodies.

## MeasurementTools

### class `MeasurementTools(grpc_client: ansys.geometry.core.connection.GrpcClient)`

Measurement tools for PyAnsys Geometry.

#### Parameters

##### `grpc_client`

[*GrpcClient*] gRPC client to use for the measurement tools.

## Overview

## Methods

<code>min_distance_between_objects</code>	Find the gap between two bodies.
---	----------------------------------

## Import detail

```
from ansys.geometry.core.tools.measurement_tools import MeasurementTools
```

## Method detail

### `MeasurementTools.min_distance_between_objects(body1: ansys.geometry.core.designer.body.Body, body2: ansys.geometry.core.designer.body.Body) → Gap`

Find the gap between two bodies.

#### Parameters

##### `body1`

[*Body*] First body to measure the gap.

##### `body2`

[*Body*] Second body to measure the gap.

#### Returns

##### `Gap`

Gap between two bodies.

## Description

Provides tools for measurement.

### The `problem_areas.py` module

#### Summary

#### Classes

<code>ProblemArea</code>	Represents problem areas.
<code>DuplicateFaceProblemAreas</code>	Provides duplicate face problem area definition.
<code>MissingFaceProblemAreas</code>	Provides missing face problem area definition.
<code>InexactEdgeProblemAreas</code>	Represents an inexact edge problem area with unique identifier and associated edges.
<code>ExtraEdgeProblemAreas</code>	Represents a extra edge problem area with unique identifier and associated edges.
<code>SmallFaceProblemAreas</code>	Represents a small face problem area with unique identifier and associated faces.
<code>SplitEdgeProblemAreas</code>	Represents a split edge problem area with unique identifier and associated edges.
<code>StitchFaceProblemAreas</code>	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.

##### Parameters

###### `id`

[str] Server-defined ID for the problem area.

###### `grpc_client`

[`GrpcClient`] Active supporting geometry service instance for design modeling.

#### Overview

#### Abstract methods

<code>fix</code>	Fix problem area.
------------------	-------------------

## Properties

<i>id</i>	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, grpc_client: ansys.geometry.core.connection.GrpcClient, faces: beartype.typing.List[ansys.geometry.core.designer.face.Face])
```

Bases: *ProblemArea*

Provides duplicate face problem area definition.

Represents a duplicate face problem area with unique identifier and associated faces.

### Parameters

**id**

[str] Server-defined ID for the body.

**grpc\_client**

[GrpcClient] Active supporting geometry service instance for design modeling.

**faces**

[List[Face]] List of faces associated with the design.

## Overview

## Methods

<i>fix</i>	Fix the problem area.
------------	-----------------------

## Properties

<code>faces</code>	The list of the edges connected to this problem area.
--------------------	---

## Import detail

```
from ansys.geometry.core.tools.problem_areas import DuplicateFaceProblemAreas
```

## Property detail

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

The list of the edges connected to this problem area.

## 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, grpc_client: ansys.geometry.core.connection.GrpcClient, edges: beartype.typing.List[ansys.geometry.core.designer.edge.Edge])`

Bases: `ProblemArea`

Provides missing face problem area definition.

### Parameters

**id**

[`str`] Server-defined ID for the body.

**grpc\_client**

[`GrpcClient`] Active supporting geometry service instance for design modeling.

**edges**

[`List[Edge]`] List of edges associated with the design.

## Overview

### Methods

<code>fix</code>	Fix the problem area.
------------------	-----------------------

### Properties

<code>edges</code>	The list 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[ansys.geometry.core.designer.edge.Edge]`

The list 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, grpc_client: ansys.geometry.core.connection.GrpcClient, edges: beartype.typing.List[ansys.geometry.core.designer.edge.Edge])`

Bases: `ProblemArea`

Represents an inexact edge problem area with unique identifier and associated edges.

#### Parameters

`id`

[`str`] Server-defined ID for the body.

`grpc_client`

[`GrpcClient`] Active supporting geometry service instance for design modeling.

`edges`

[`List[Edge]`] List of edges associated with the design.

## Overview

### Methods

<code>fix</code>	Fix the problem area.
------------------	-----------------------

### Properties

<code>edges</code>	The list 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[ansys.geometry.core.designer.edge.Edge]`

The list 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, grpc_client: ansys.geometry.core.connection.GrpcClient, edges:
                            beartype.typing.List[ansys.geometry.core.designer.edge.Edge])
```

Bases: `ProblemArea`

Represents a extra edge problem area with unique identifier and associated edges.

#### Parameters

`id`

[`str`] Server-defined ID for the body.

`grpc_client`

[`GrpcClient`] Active supporting geometry service instance for design modeling.

`edges`

[`List[Edge]`] List of edges associated with the design.

## Overview

### Properties

<code>edges</code>	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[ansys.geometry.core.designer.edge.Edge]
```

The list of the ids of the edges connected to this problem area.

## SmallFaceProblemAreas

```
class SmallFaceProblemAreas(id: str, grpc_client: ansys.geometry.core.connection.GrpcClient, faces:  
beartype.typing.List[ansys.geometry.core.designer.face.Face])
```

Bases: *ProblemArea*

Represents a small face problem area with unique identifier and associated faces.

#### Parameters

##### `id`

[`str`] Server-defined ID for the body.

##### `grpc_client`

[`GrpcClient`] Active supporting geometry service instance for design modeling.

##### `faces`

[`List[Face]`] List of edges associated with the design.

## Overview

### Methods

<code>fix</code>	Fix the problem area.
------------------	-----------------------

## Properties

<code>faces</code>	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[ansys.geometry.core.designer.face.Face]`

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, grpc_client: ansys.geometry.core.connection.GrpcClient, edges: beartype.typing.List[ansys.geometry.core.designer.edge.Edge])`

Bases: `ProblemArea`

Represents a split edge problem area with unique identifier and associated edges.

### Parameters

`id`

[`str`] Server-defined ID for the body.

`grpc_client`

[`GrpcClient`] Active supporting geometry service instance for design modeling.

`edges`

[`List[Edge]`] List of edges associated with the design.

## Overview

### Methods

<code>fix</code>	Fix the problem area.
------------------	-----------------------

### Properties

<code>edges</code>	The list of 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[ansys.geometry.core.designer.edge.Edge]`

The list of 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, grpc_client: ansys.geometry.core.connection.GrpcClient, bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body])`

Bases: `ProblemArea`

Represents a stitch face problem area with unique identifier and associated faces.

#### Parameters

`id`

[`str`] Server-defined ID for the body.

`grpc_client`

[`GrpcClient`] Active supporting geometry service instance for design modeling.

`bodies`

[`List[Body]`] List of bodies associated with the design.

## Overview

### Methods

<i>fix</i>	Fix the problem area.
------------	-----------------------

### Properties

<i>bodies</i>	The list of the bodies connected to this problem area.
---------------	--

### Import detail

```
from ansys.geometry.core.tools.problem_areas import StitchFaceProblemAreas
```

### Property detail

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

The list of the bodies 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

<i>RepairToolMessage</i>	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

<code>success</code>	The success of the repair operation.
<code>created_bodies</code>	The list of the created bodies after the repair operation.
<code>modified_bodies</code>	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

<code>find_split_edges</code>	Find split edges in the given list of bodies.
<code>find_extra_edges</code>	Find the extra edges in the given list of bodies.
<code>find_inexact_edges</code>	Find inexact edges in the given list of bodies.
<code>find_duplicate_faces</code>	Find the duplicate face problem areas.
<code>find_missing_faces</code>	Find the missing faces.
<code>find_small_faces</code>	Find the small face problem areas.
<code>find_stitch_faces</code>	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(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body], 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

###### bodies

[List[Body]] List of bodies that split edges are investigated on.

###### 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(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body]) → 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

### bodies

[List[Body]] List of bodies that extra edges are investigated on.

## Returns

### List[ExtraEdgeProblemArea]

List of objects representing extra edge problem areas.

RepairTools.**find\_inexact\_edges**(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body]) →  
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

### bodies

[List[Body]] List of bodies that inexact edges are investigated on.

## Returns

### List[InExactEdgeProblemArea]

List of objects representing inexact edge problem areas.

RepairTools.**find\_duplicate\_faces**(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body])  
→  
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

### bodies

[List[Body]] List of bodies that duplicate faces are investigated on.

## Returns

### List[DuplicateFaceProblemAreas]

List of objects representing duplicate face problem areas.

RepairTools.**find\_missing\_faces**(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body]) →  
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

### bodies

[List[Body]] List of bodies that missing faces are investigated on.

## Returns

### List[MissingFaceProblemAreas]

List of objects representing missing face problem areas.

RepairTools.**find\_small\_faces**(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body]) →  
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

### bodies

[List[*Body*]] List of bodies that small faces are investigated on.

## Returns

### List[*SmallFaceProblemAreas*]

List of objects representing small face problem areas.

`RepairTools.find_stitch_faces(bodies: beartype.typing.List[ansys.geometry.core.designer.body.Body]) → 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

### bodies

[List[*Body*]] List of bodies that stitchable faces are investigated on.

## 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

---

<code>GeometryRuntimeError</code>	Provides error message to raise when Geometry service passes a runtime error.
<code>GeometryExitedError</code>	Provides error message to raise when Geometry service has exited.

---

## Functions

---

<code>handler</code>	Pass signal to the custom interrupt handler.
<code>protect_grpc</code>	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.

#### Parameters

`msg`  
[`str`, default: “Geometry service has exited.”] Message to raise.

#### 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

<i>PyGeometryCustomAdapter</i>	Keeps the reference to the Geometry service instance name dynamic.
<i>PyGeometryPercentStyle</i>	Provides a common messaging style for the PyGeometryFormatter class.
<i>PyGeometryFormatter</i>	Provides a Formatter class for overwriting default format styles.
<i>InstanceFilter</i>	Ensures that the <code>instance_name</code> record always exists.
<i>Logger</i>	Provides the logger used for each PyAnsys Geometry session.

### Functions

<i>addfile_handler</i>	Add a file handler to the input.
<i>add_stdout_handler</i>	Add a standout handler to the logger.

### Attributes

<i>string_to_loglevel</i>
---------------------------

### Constants

<i>LOG_LEVEL</i>
<i>FILE_NAME</i>
<i>DEBUG</i>
<i>INFO</i>
<i>WARN</i>
<i>ERROR</i>
<i>CRITICAL</i>
<i>STDOUT_MSG_FORMAT</i>
<i>FILE_MSG_FORMAT</i>
<i>DEFAULT_STDOUT_HEADER</i>
<i>DEFAULT_FILE_HEADER</i>
<i>NEW_SESSION_HEADER</i>
<i>LOG</i>

## PyGeometryCustomAdapter

```
class PyGeometryCustomAdapter(logger, extra=None)
```

Bases: `logging.LoggerAdapter`

Keeps the reference to the Geometry service instance name dynamic.

If you use the standard approach, which is supplying *extra* input to the logger, you must input Geometry service instances each time you do a log.

Using adapters, you only need to specify the Geometry service instance that you are referring to once.

## Overview

### Methods

<code>process</code>	Process the logging message and keyword arguments passed in to
<code>log_to_file</code>	Add a file handler to the logger.
<code>log_to_stdout</code>	Add a standard output handler to the logger.
<code>setLevel</code>	Change the log level of the object and the attached handlers.

### Attributes

<code>level</code>
<code>file_handler</code>
<code>stdout_handler</code>

### 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=STDOOUT_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

<code>filter</code>	Ensure that the <code>instance_name</code> 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.

This class allows you to add handlers to the logger to output messages to a file or to the standard output (stdout).

### Parameters

**level**

[`int`, default: 10] Logging level to filter the message severity allowed in the logger. The default is 10, in which case the `logging.DEBUG` level is used.

**to\_file**

[`bool`, default: `False`] Whether to write log messages to a file.

**to\_stdout**

[`bool`, default: `True`] Whether to write log messages to the standard output.

**filename**

[`str`, default: “`pyansys-geometry.log`”] Name of the file to write log messages to.

## Examples

Demonstrate logger usage from the `Modeler` instance, which is automatically created when a Geometry service instance is created.

```
>>> from ansys.geometry.core import Modeler
>>> modeler = Modeler(loglevel='DEBUG')
>>> modeler._log.info('This is a useful message')
INFO - - <ipython-input-24-80df150fe31f> - <module> - This is LOG debug message.
```

Import the global PyAnsys Geometry logger and add a file output handler.

```
>>> import os
>>> from ansys.geometry.core import LOG
>>> file_path = os.path.join(os.getcwd(), 'pyansys-geometry.log')
>>> LOG.log_to_file(file_path)
```

## Overview

### Methods

<code>log_to_file</code>	Add a file handler to the logger.
<code>log_to_stdout</code>	Add the standard output handler to the logger.
<code>setLevel</code>	Change the log level of the object and the attached handlers.
<code>add_child_logger</code>	Add a child logger to the main logger.
<code>add_instance_logger</code>	Add a logger for a Geometry service instance.
<code>add_handling_uncaught_exceptions</code>	Redirect the output of an exception to a logger.

### Attributes

<code>file_handler</code>
<code>std_out_handler</code>

## Special methods

<code>__getitem__</code>	Overload the access method by item for the <code>Logger</code> 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(suffix: 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

##### `suffix`

[`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_exceptions(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

<code>Modeler</code>	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.platforminstancemanagement.Instance] = None, docker_instance: beartype.typing.Optional[ansys.geometry.core.connection.docker_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.BackendType] = None)
```

Provides for interacting with an open session of the Geometry service.

### Parameters

#### host

[str, default: DEFAULT\_HOST] Host where the server is running.

#### port

[Union[str, int], default: DEFAULT\_PORT] Port number where the server is running.

#### channel

[Channel, default: None] gRPC channel for server communication.

#### remote\_instance

[ansys.platforminstancemanagement.Instance, default: None] Corresponding remote instance when the Geometry service is launched using PyPIM. This instance is deleted when the GrpcClient.close method is called.

#### docker\_instance

[LocalDockerInstance, default: None] Corresponding local Docker instance when the Geometry service is launched using the launch\_docker\_modeler method. This instance is deleted when the GrpcClient.close method is called.

#### product\_instance

[ProductInstance, default: None] Corresponding local product instance when the product (Discovery or SpaceClaim) is launched through the launch\_modeler\_with\_geometry\_service(), launch\_modeler\_with\_discovery() or the launch\_modeler\_with\_spaceclaim() interface. This instance will be deleted when the GrpcClient.close method is called.

#### timeout

[Real, default: 120] Time in seconds for trying to achieve the connection.

#### logging\_level

[int, default: INFO] Logging level to apply to the client.

#### logging\_file

[str, Path, default: None] File to output the log to, if requested.

## Overview

### Methods

<code>create_design</code>	Initialize a new design with the connected client.
<code>get_active_design</code>	Get the active design on the modeler object.
<code>read_existing_design</code>	Read the existing design on the service with the connected client.
<code>close</code>	Modeler method for easily accessing the client's close method.
<code>open_file</code>	Open a file.
<code>run_discovery_script_file</code>	Run a Discovery script file.

### Properties

<code>client</code>	Modeler instance client.
<code>repair_tools</code>	Access to repair tools.
<code>measurement_tools</code>	Access to measurement tools.

### Special methods

<code>__repr__</code>	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.

property Modeler.measurement_tools: MeasurementTools
    Access to measurement 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.get_active_design(sync_with_backend: bool = True) → ansys.geometry.core.designer.design.Design`

Get the active design on the modeler object.

### Parameters

#### `sync_with_backend`

[`bool`, default: `True`] Whether to sync the active design with the remote service. If set to `False`, the active design may be out-of-sync with the remote service. This is useful when the active design is known to be up-to-date.

### Returns

#### `Design`

Design object already existing on the modeler.

`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.Optional[beartype.typing.Dict[str, str]] = None,  
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.

---

**Note:** If arguments are passed to the script, they must be in the form of a dictionary. On the server side, the script will receive the arguments as a dictionary of strings, under the variable name argsDict. For example, if the script is called with the arguments run\_discovery\_script\_file(..., script\_args = {"length": "20"}, ...), the script will receive the dictionary argsDict with the key-value pair {"length": "20"}.

---

---

**Note:** If an output is expected from the script, it will be returned as a dictionary of strings. The keys and values of the dictionary are the variables and their values that the script returns. However, it is necessary that the script creates a dictionary called result with the variables and their values that are expected to be returned. For example, if the script is expected to return the number of bodies in the design, the script should create a dictionary called result with the key-value pair {"numBodies": numBodies}, where numBodies is the number of bodies in the design.

---

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**

[*Optional[Dict[str, str]]*, optional.] Arguments to pass to the script. By default, None.

**import\_design**

[*bool*, optional.] 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. By default, False.

**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

<code>Real</code>	Type used to refer to both integers and floats as possible values.
<code>RealSequence</code>	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.ndarray`s 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_TNAME: bool = False`

Global constant for checking whether to use `tname` for visualization.

`core.DISABLE_MULTIPLE_DESIGN_CHECK: bool = False`

Global constant for disabling the `ensure_design_is_active` check.

Only set this to false if you are sure you want to disable this check and you will ONLY be working with one design.

`core.DOCUMENTATION_BUILD: bool`

Global flag to set when building the documentation to use the proper PyVista Jupyter backend.

`core.__version__`

PyAnsys Geometry version.

## 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]: 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(v2))

print("\nVectors are parallel:")
print(v1.is_parallel(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]: 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]: 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]: 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]: 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

PyAnsyst 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]: 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]: 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]: 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:

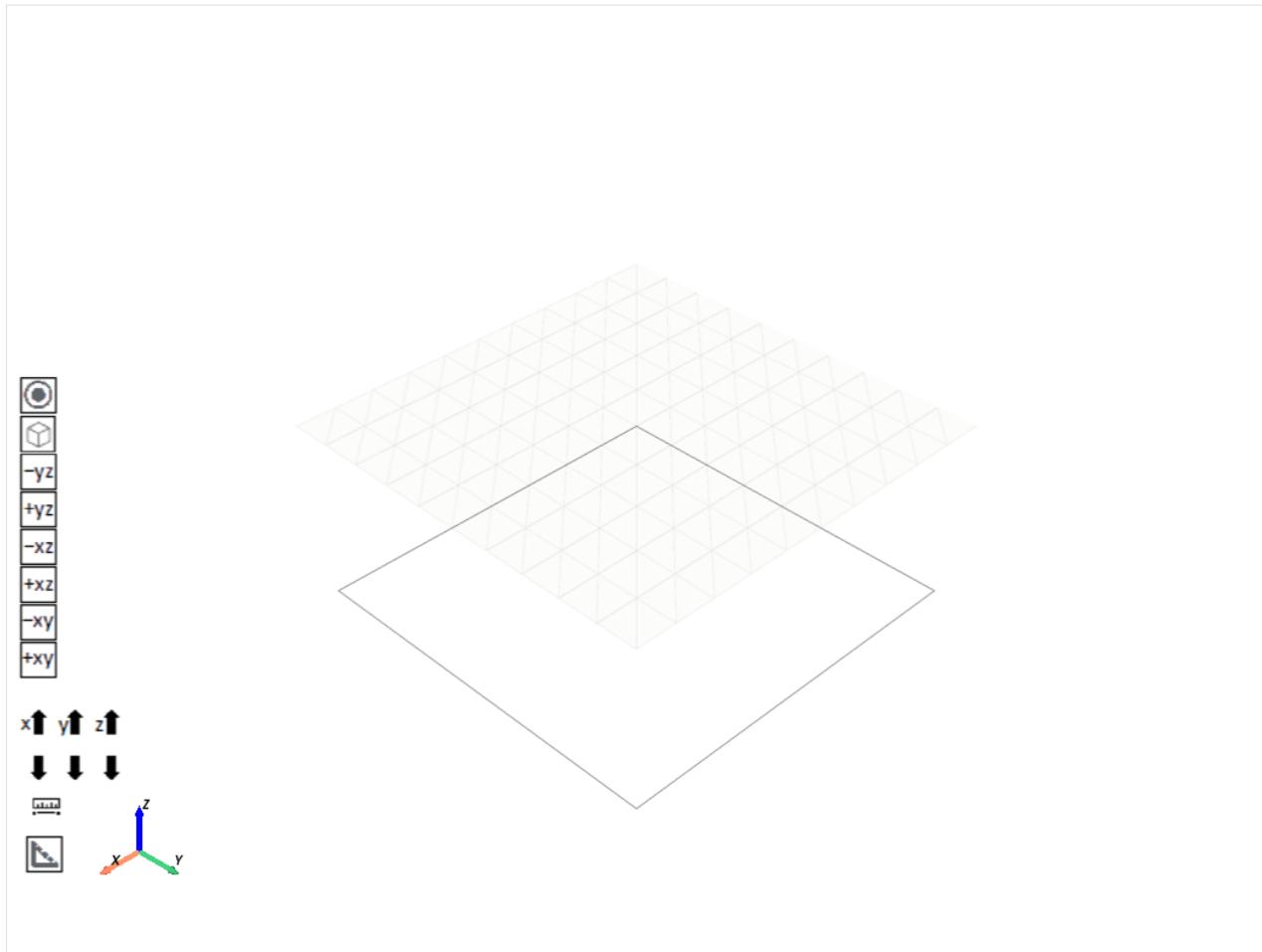
- arc
- box
- circle
- ellipse
- gear
- polygon
- segment
- slot
- trapezoid
- triangle

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]))

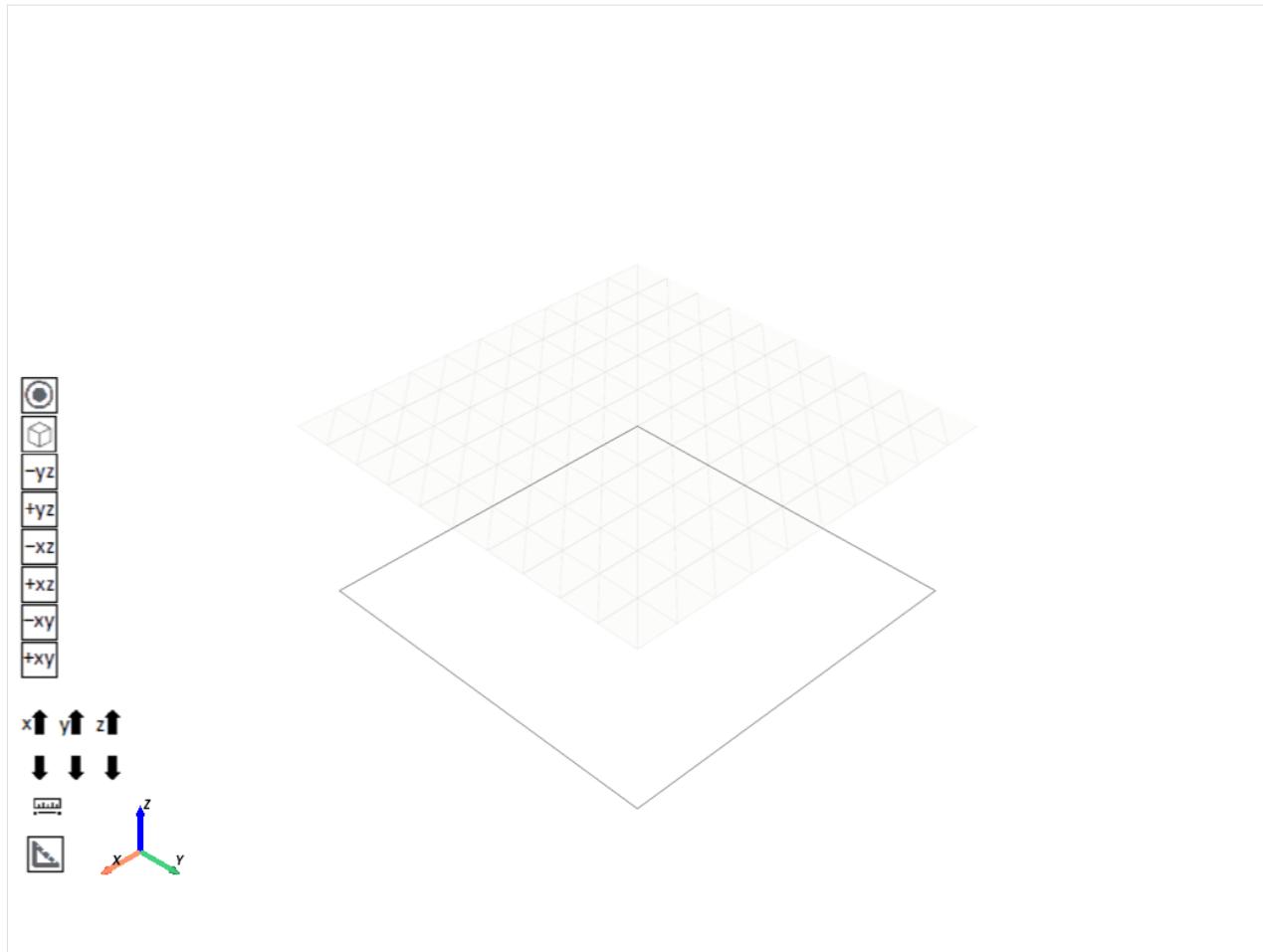
sketch.plot()
```



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.

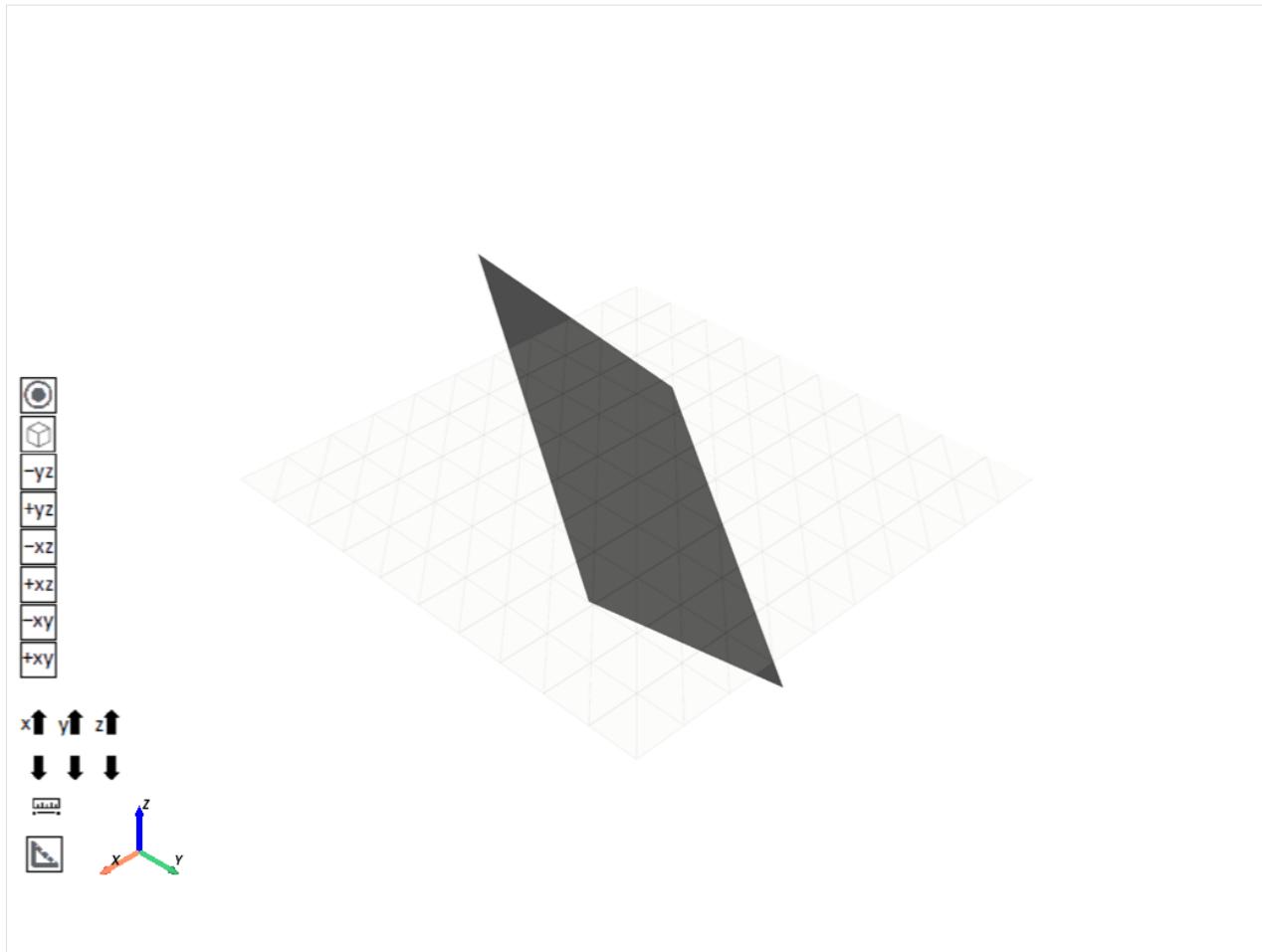
```
[3]: sketch = Sketch()
(
    sketch.segment(Point2D([0,0]), Point2D([0,1]))
        .segment_to_point(Point2D([1,1]))
        .segment_to_point(Point2D([1,0]))
        .segment_to_point(Point2D([0,0]))
)
sketch.plot()
```



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()
```



## Combine concepts to create powerful sketches

Combine these simple concepts to create powerful sketches.

```
[5]: # Complex Fluent API Sketch - PCB

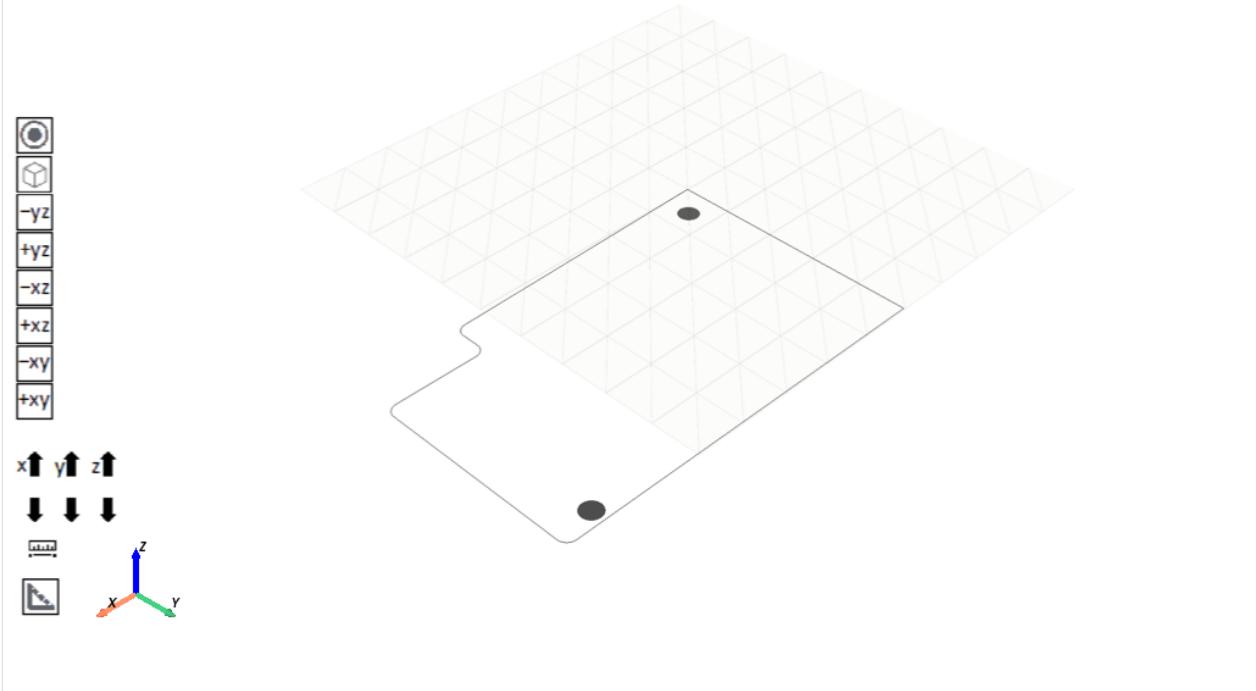
sketch = Sketch()

(
    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))
```

(continues on next page)

(continued from previous page)

```
.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()
```



---

**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 (0x281829c9be0)

Ansys Geometry Modeler Client (0x281829c9b80)
  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.

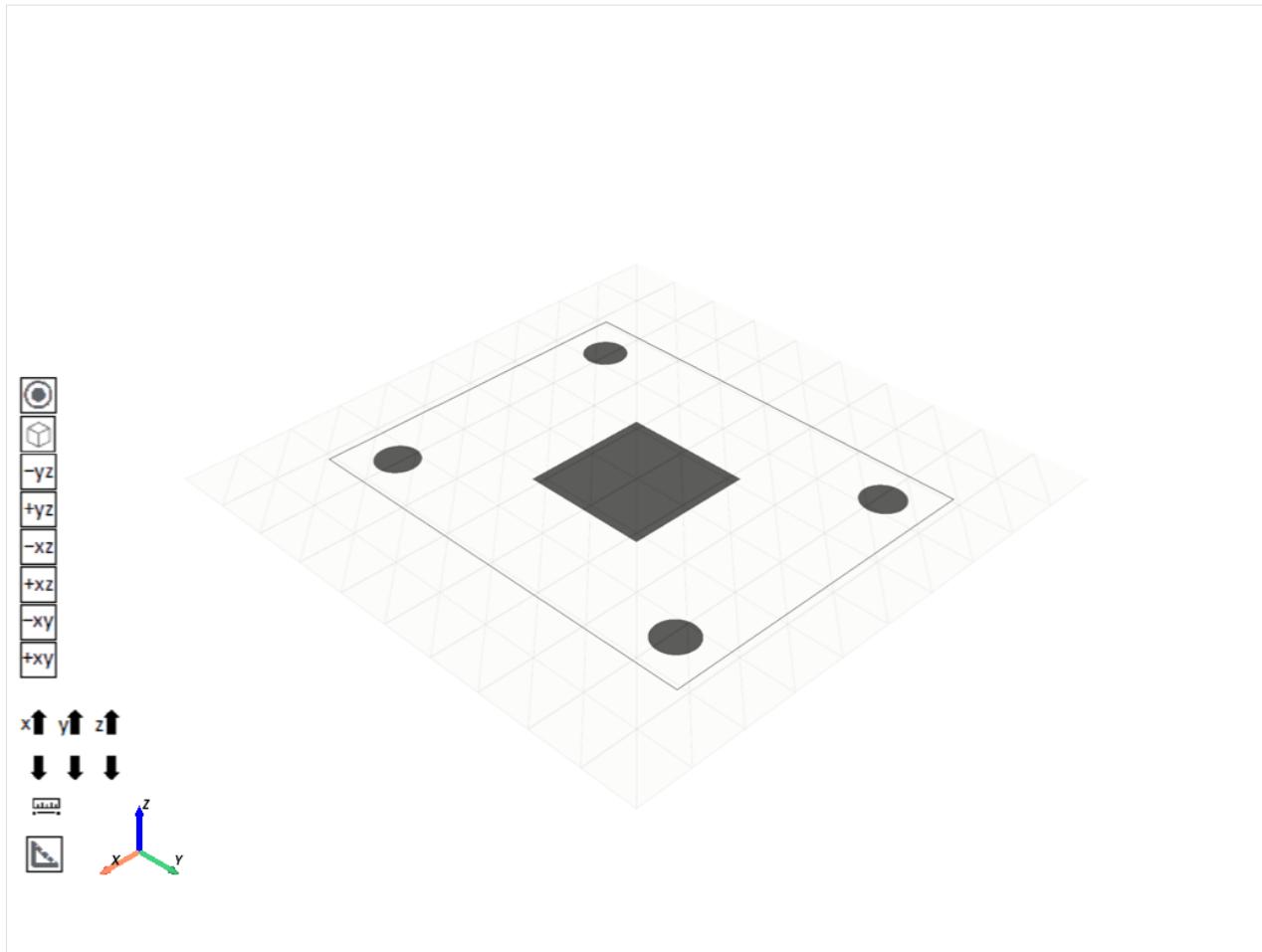
## Create and plot a sketch

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))
    .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()
```



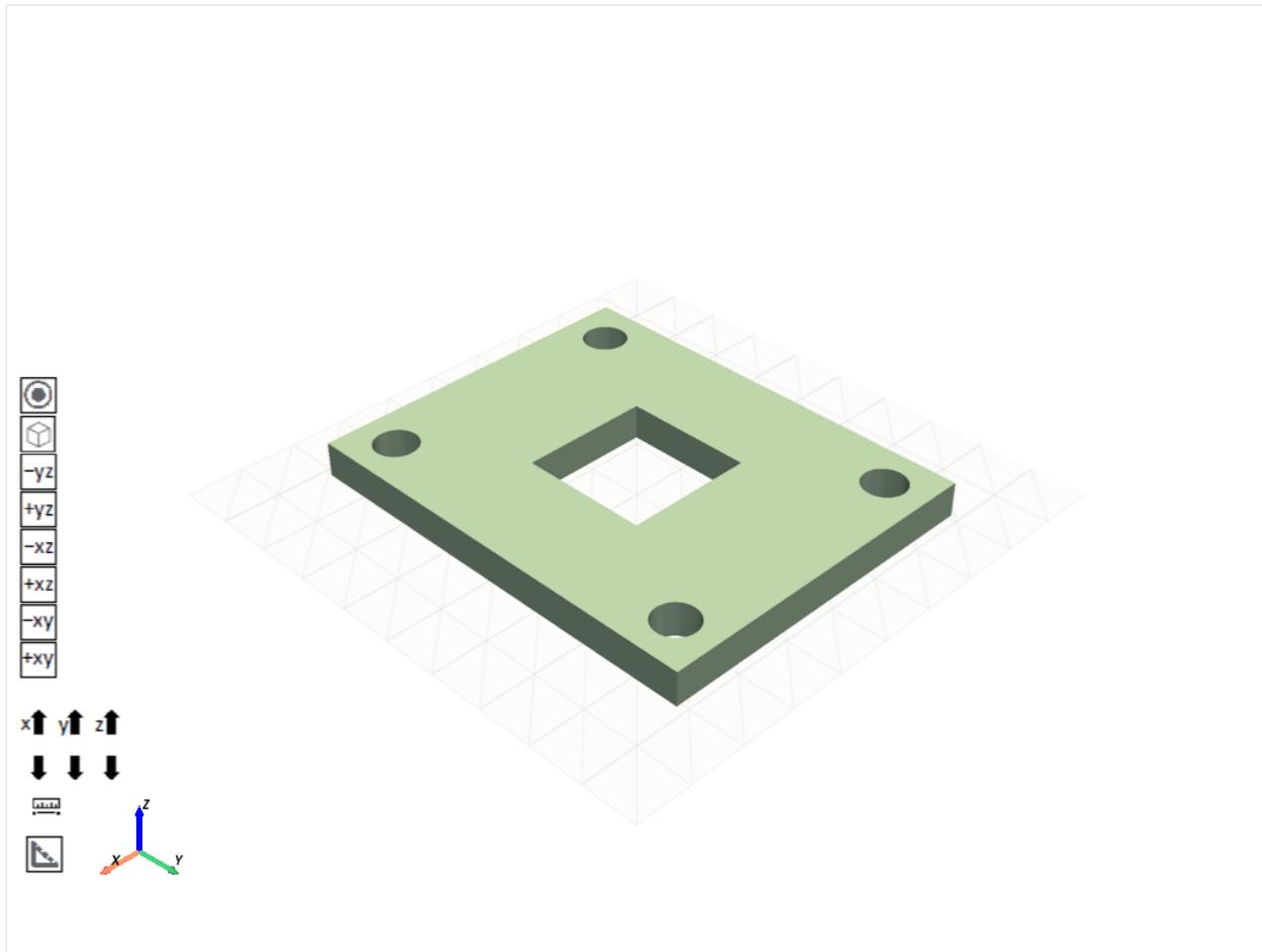
### Perform some modeling operations

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()
```



### 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.

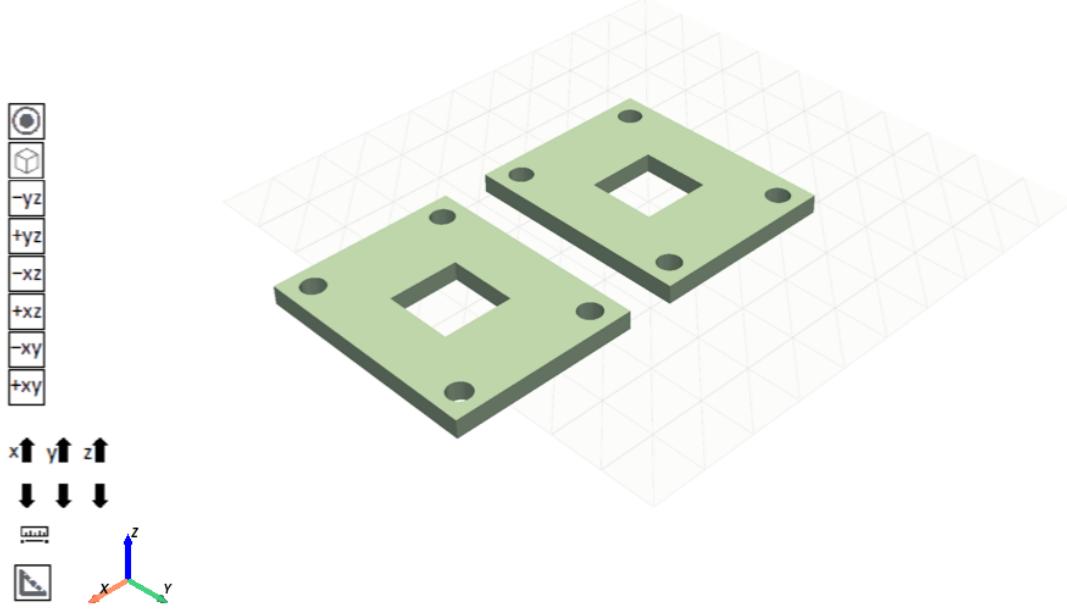
```
[5]: from ansys.geometry.core.math import UNITVECTOR3D_X

# 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()
```



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))
```

(continues on next page)

(continued from previous page)

```

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",
    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

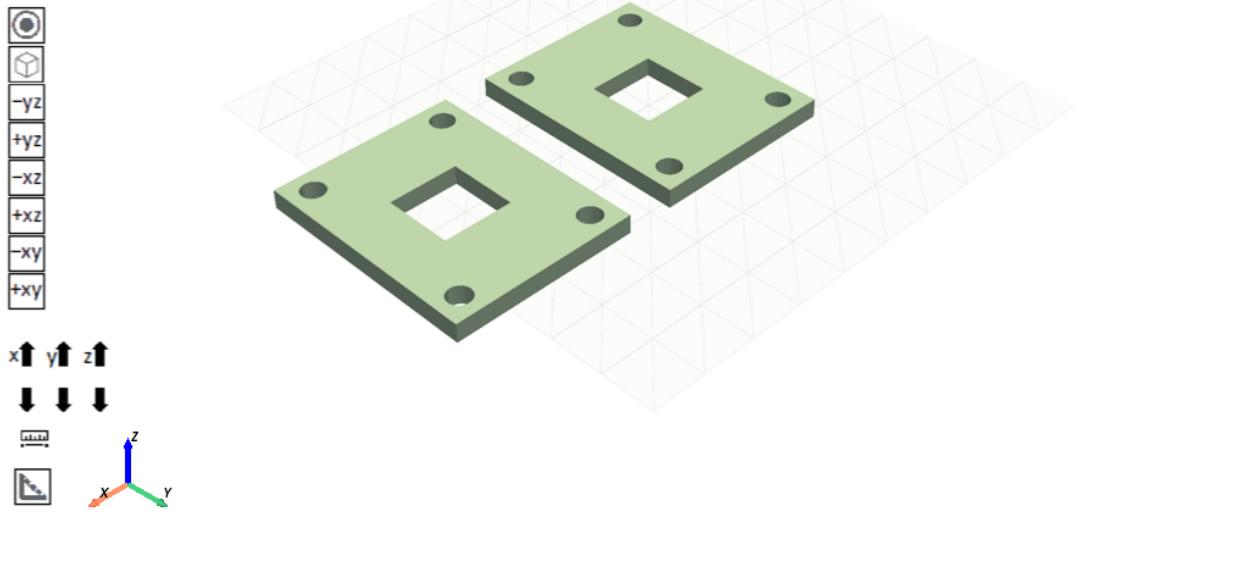
Deletion operations for bodies, named selections, and components are possible, always from the scope expected. For example, if you attempted to delete the original body from a component that has no ownership over it (such as your `comp` object), the deletion would fail. If you attempted to perform this 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}")

# If you request a plot of the entire design, you can still see it.
design.plot()

Is the body alive? True
```



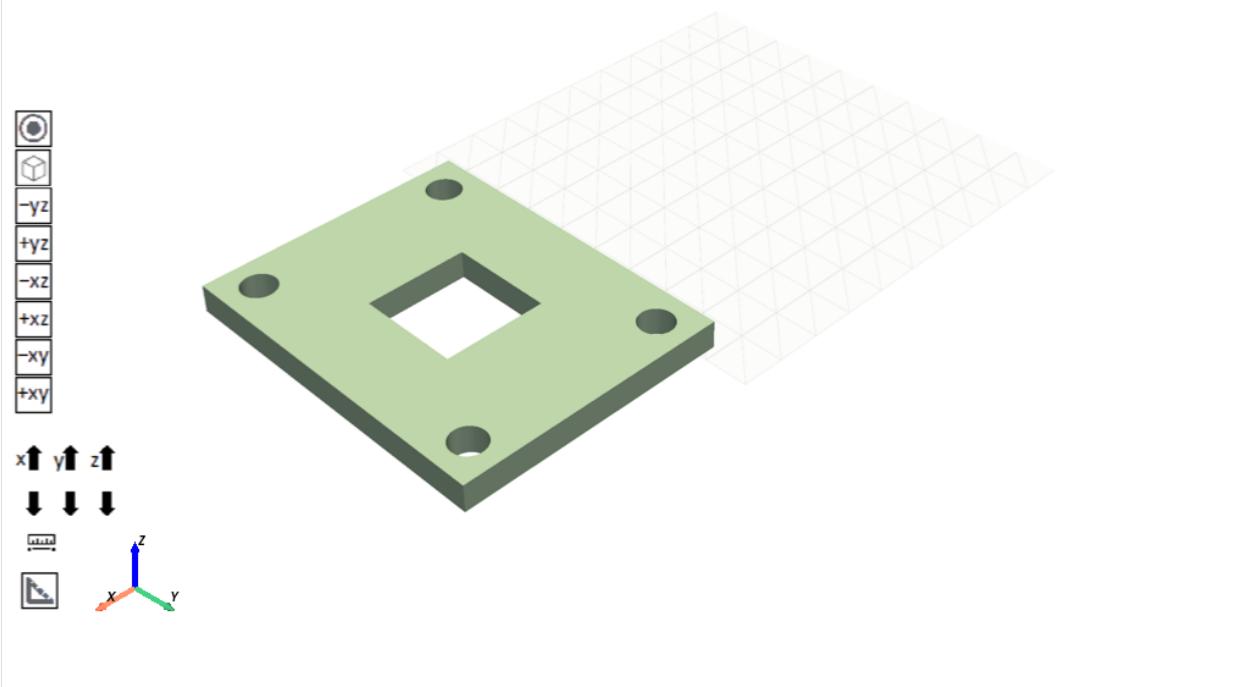
```
[9]: # Because the body belongs to the ``design`` object and not the ``comp`` object,
# deleting it from ``design`` object works.
design.delete_body(body)
print(f"Is the body alive? {body.is_alive}")
```

(continues on next page)

(continued from previous page)

```
# If you request a plot of the entire design, it is no longer visible.
design.plot()

Is the body alive? False
```



## Export files

Once modeling operations are finalized, you can export files in different formats. For the formats supported by DMS, see the [DesignFormat](#) 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 DesignFormat

# Path to downloads directory
file_dir = Path(os.getcwd(), "downloads")
file_dir.mkdir(parents=True, exist_ok=True)

# Download the model in different formats
```

(continues on next page)

(continued from previous page)

```
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 session 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.docker_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

docker_instance = LocalDockerInstance(
    connect_to_existing_service=True,
    restart_if_existing_service=True,
    image=is_image_available_cont,
)
modeler = Modeler(docker_instance=docker_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.

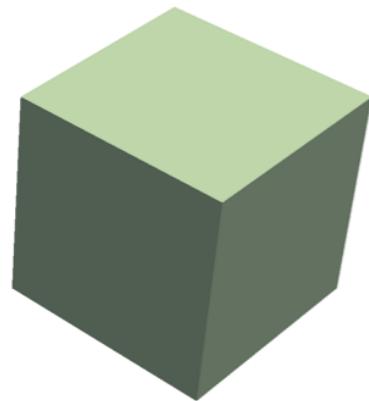
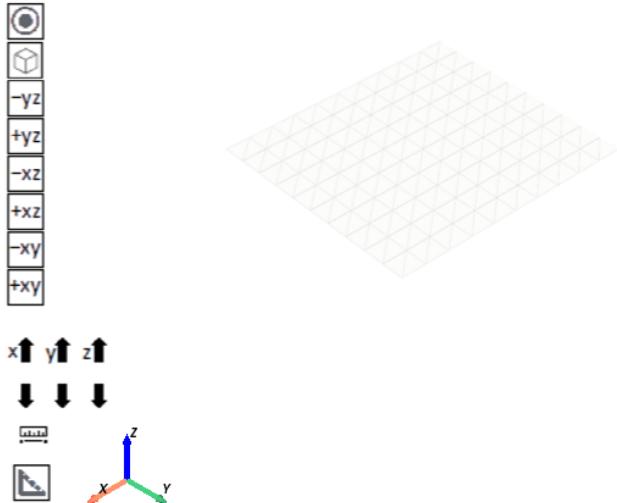
```
[8]: box2 = Sketch()
box2.box(Point2D([-10, 20], UNITS.m), Quantity(10, UNITS.m), Quantity(10, UNITS.m))
box_body2 = design.extrude_sketch("JustABox", box2, Quantity(10, UNITS.m))
plot_list.append(box_body2)
```

## 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)
```



```
[9]: []
```

You can plot the whole list of objects.

```
[10]: plotter = PlotterHelper()  
plotter.plot(plot_list)
```



[10]: []

The Python visualizer is used by default. However, you can also use `trame` for visualization.

```
plotter = PlotterHelper(use_trame=True)
plotter.plot(plot_list)
```

## Clip objects

You can clip any object represented in the plotter by defining a `Plane` object that intersects the target object.

```
[11]: from ansys.geometry.core.math import Plane, Point3D
ph = PlotterHelper()

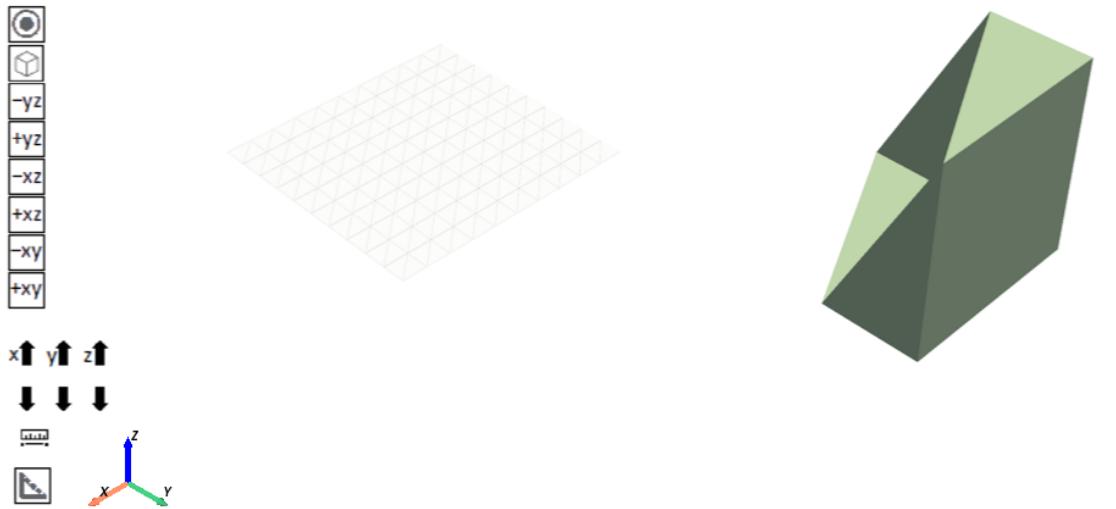
# Define PyAnsys Geometry box
box2 = Sketch()
box2.box(Point2D([-10, 20], UNITS.m), Quantity(10, UNITS.m), Quantity(10, UNITS.m))
box_body2 = design.extrude_sketch("JustABox", box2, Quantity(10, UNITS.m))

# Define plane to clip the box
origin = Point3D([-10., 20., 5.], UNITS.m)
plane = Plane(origin=origin, direction_x=[1, 1, 1], direction_y=[-1, 0, 1])
```

(continues on next page)

(continued from previous page)

```
# Add the object with the clipping plane
ph.add(box_body2, clipping_plane=plane)
ph.plot()
```



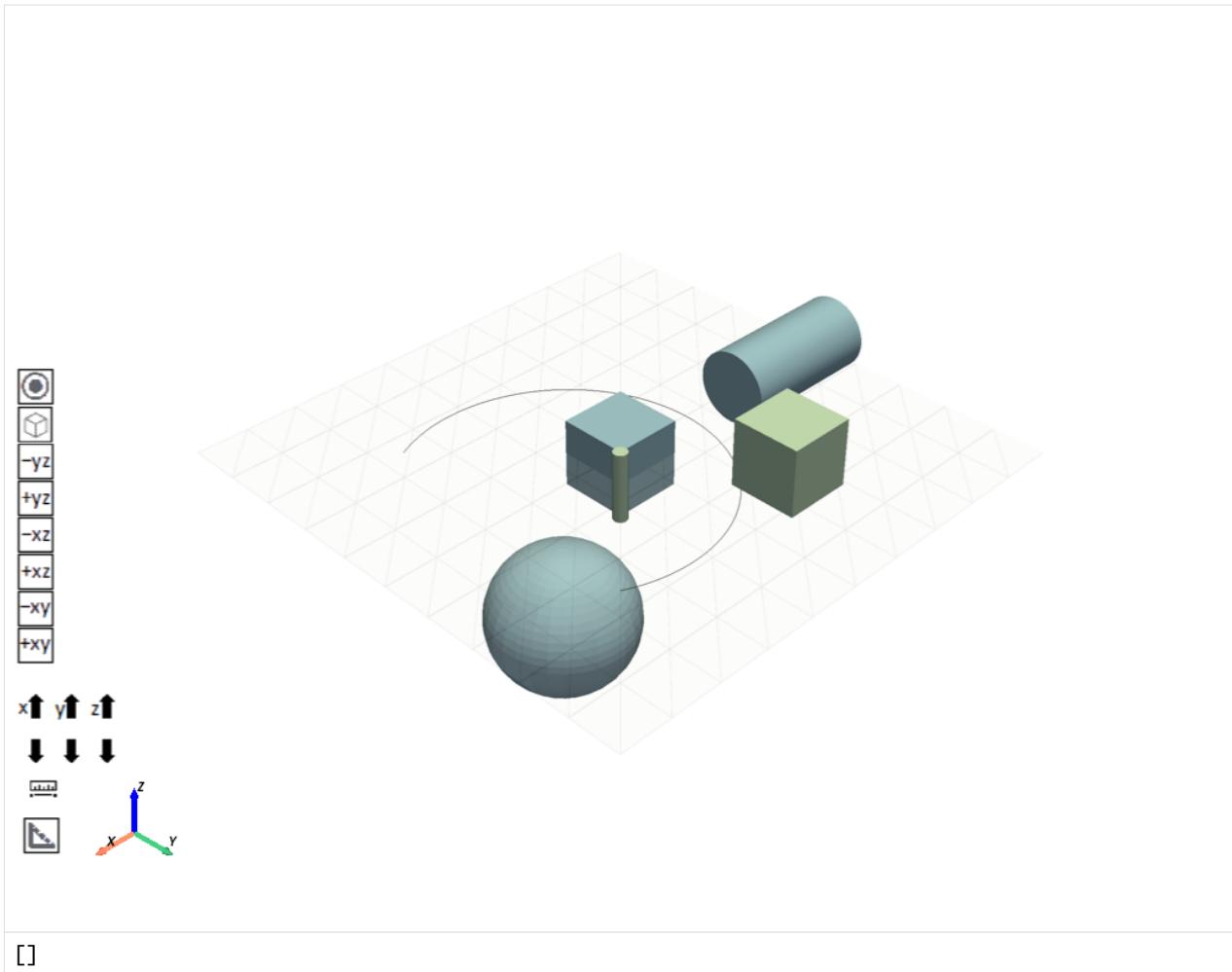
```
[11]: []
```

### Select objects interactively

PyAnsys Geometry's plotter supports interactive object selection within the scene. This enables you to pick objects for subsequent script manipulation.

```
[12]: plotter = PlotterHelper(allow_picking=True)

# Plotter returns picked bodies
picked_list = plotter.plot(plot_list)
print(picked_list)
```



## 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.

```
[13]: 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)
```

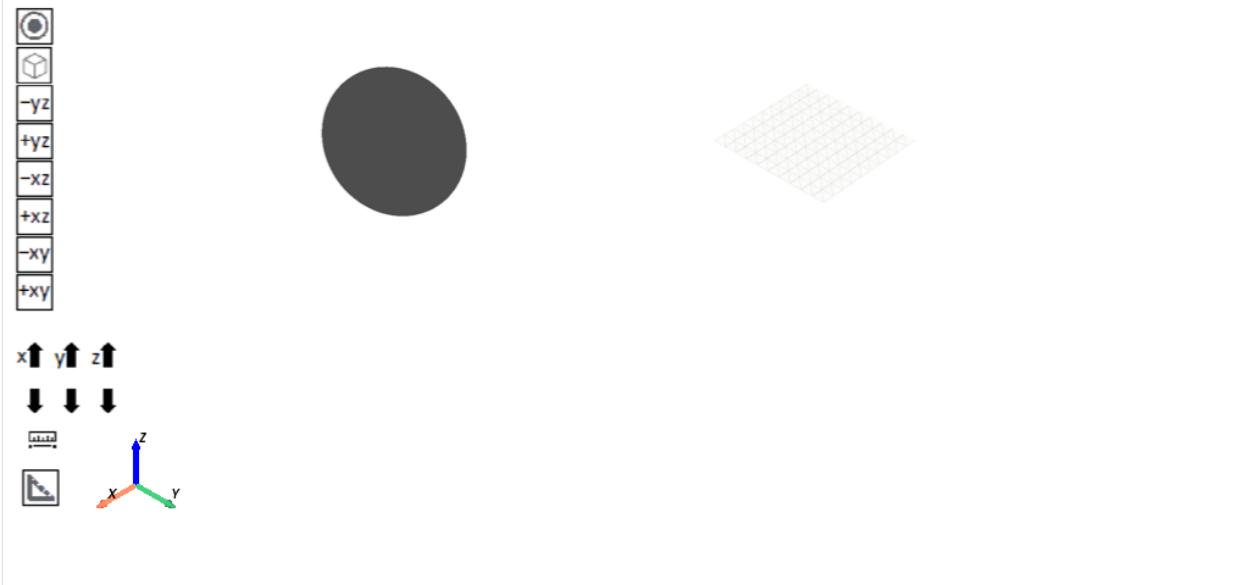
## Draw shapes

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.

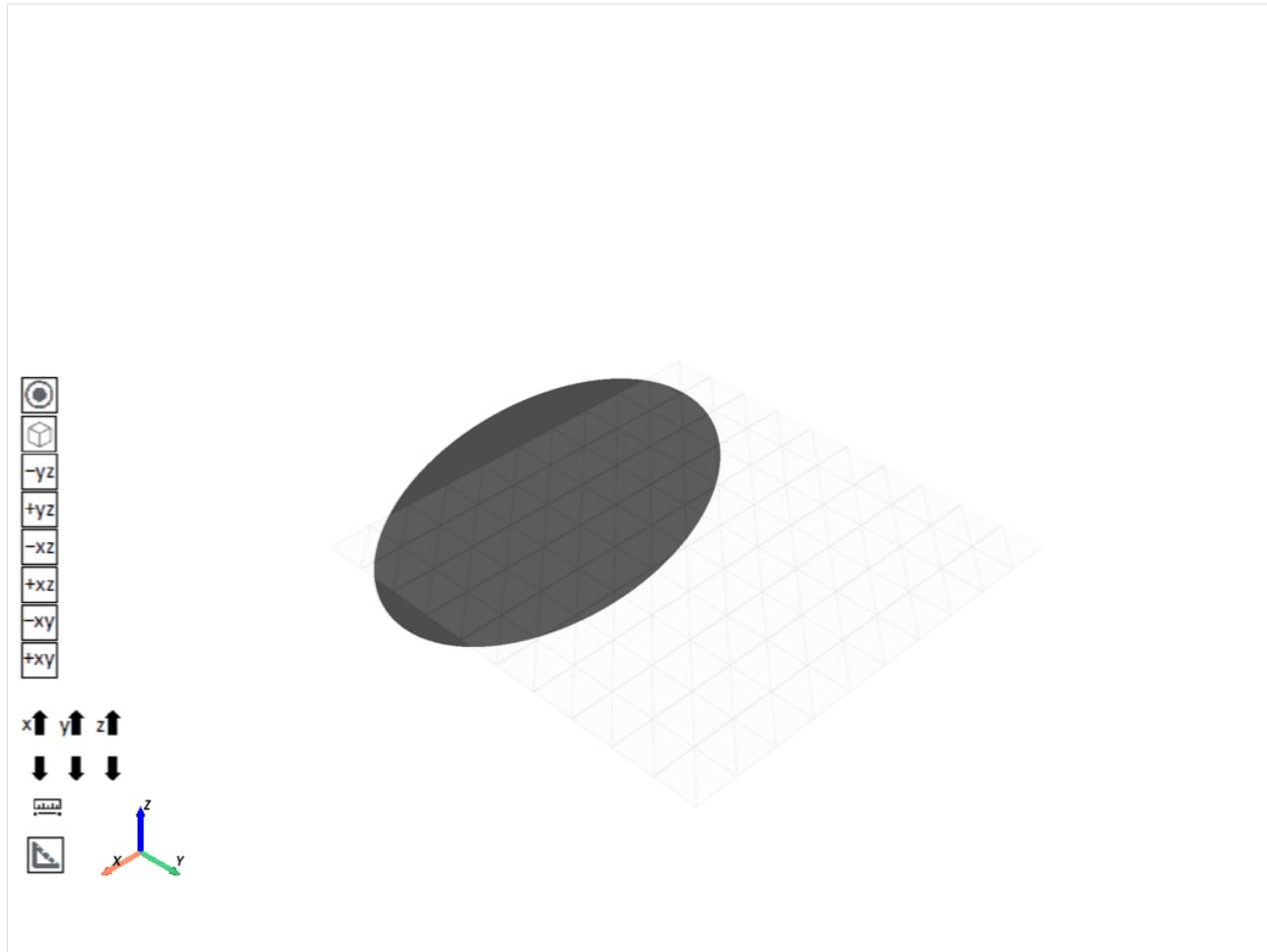
```
[4]: sketch.circle(Point2D([2, 1]), radius=30 * u.cm, tag="Circle")
sketch.select("Circle")
sketch.plot_selection()
```



### Draw an ellipse

You draw an ellipse in a sketch by specifying the center, major radius, and minor radius.

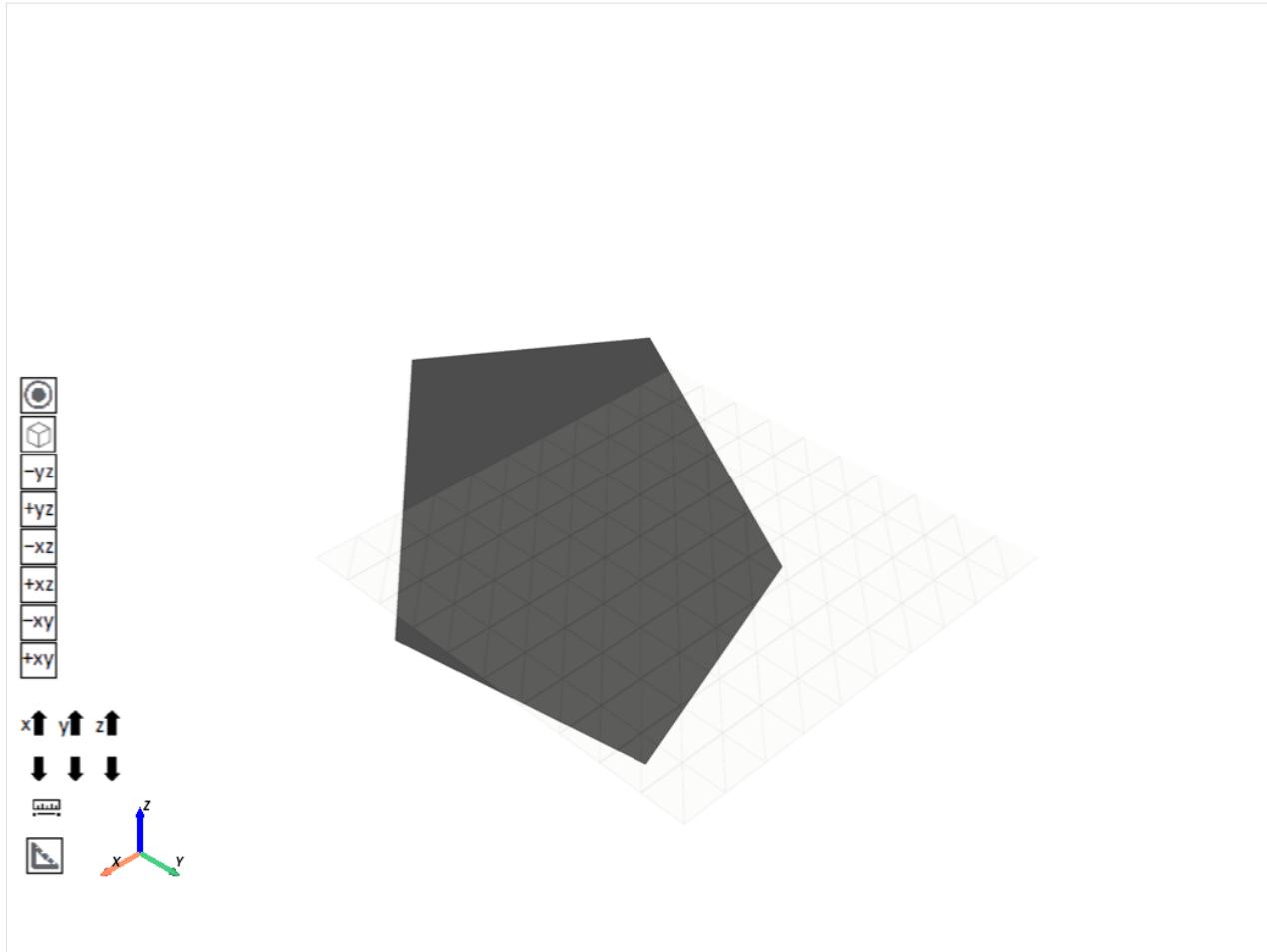
```
[5]: sketch.ellipse(
    Point2D([1, 1]), major_radius=2*u.m, minor_radius=1*u.m, tag="Ellipse"
)
sketch.select("Ellipse")
sketch.plot_selection()
```



## Draw a polygon

You draw a regular polygon by specifying the center, radius, and desired number of sides.

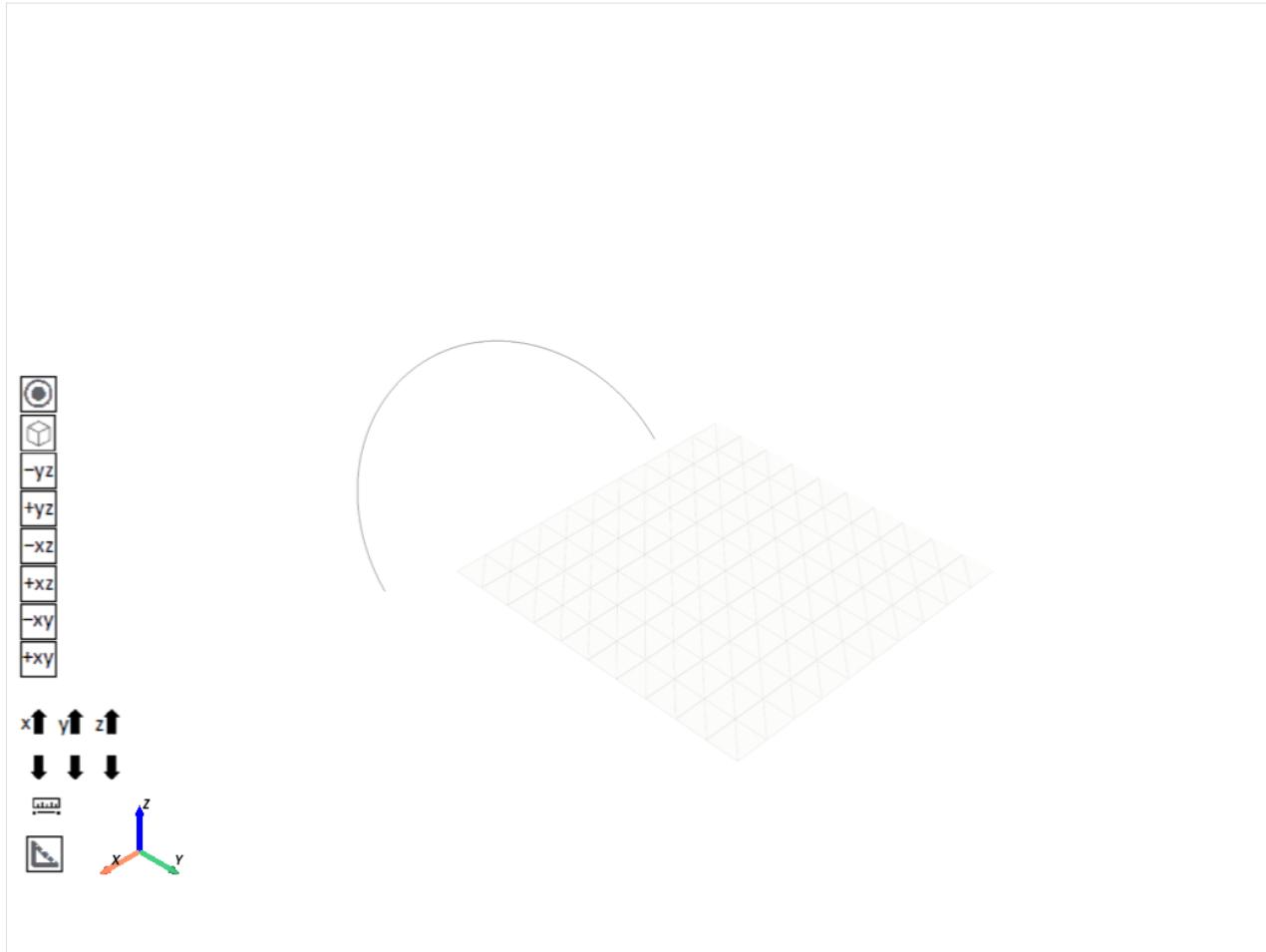
```
[6]: sketch.polygon(  
    Point2D([1, 1]), inner_radius=3*u.m, sides=5, tag="Polygon"  
)  
sketch.select("Polygon")  
sketch.plot_selection()
```



## Draw an arc

You draw an arc of circumference by specifying the center, starting point, and ending point.

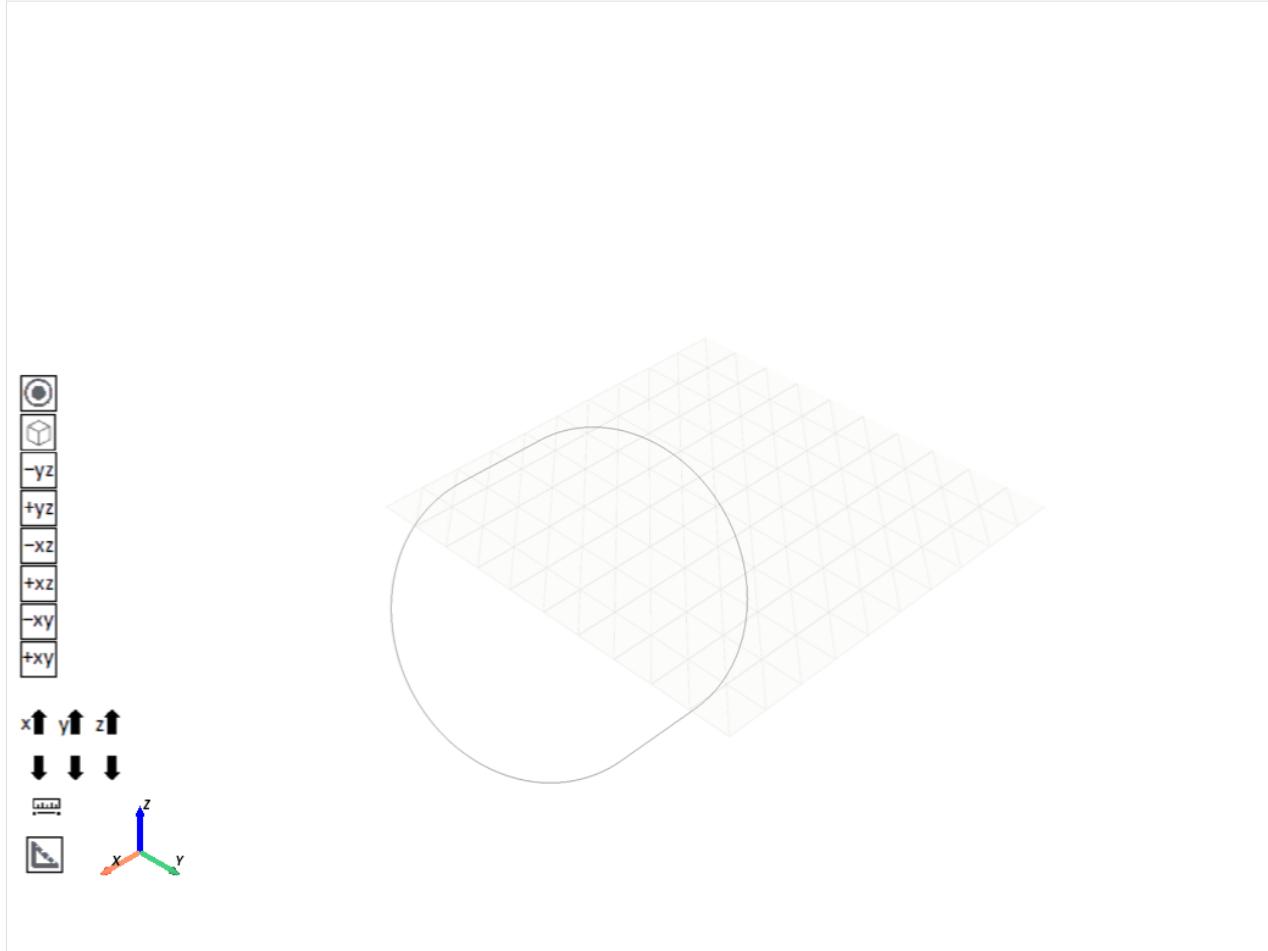
```
[7]: start_point, end_point = Point2D([2, 1], unit=u.m), Point2D([0, 1], unit=u.meter)
sketch.arc(start_point, end_point, Point2D([1, 1]), tag="Arc")
sketch.select("Arc")
sketch.plot_selection()
```



## Draw a slot

You draw a slot by specifying the center, width, and height.

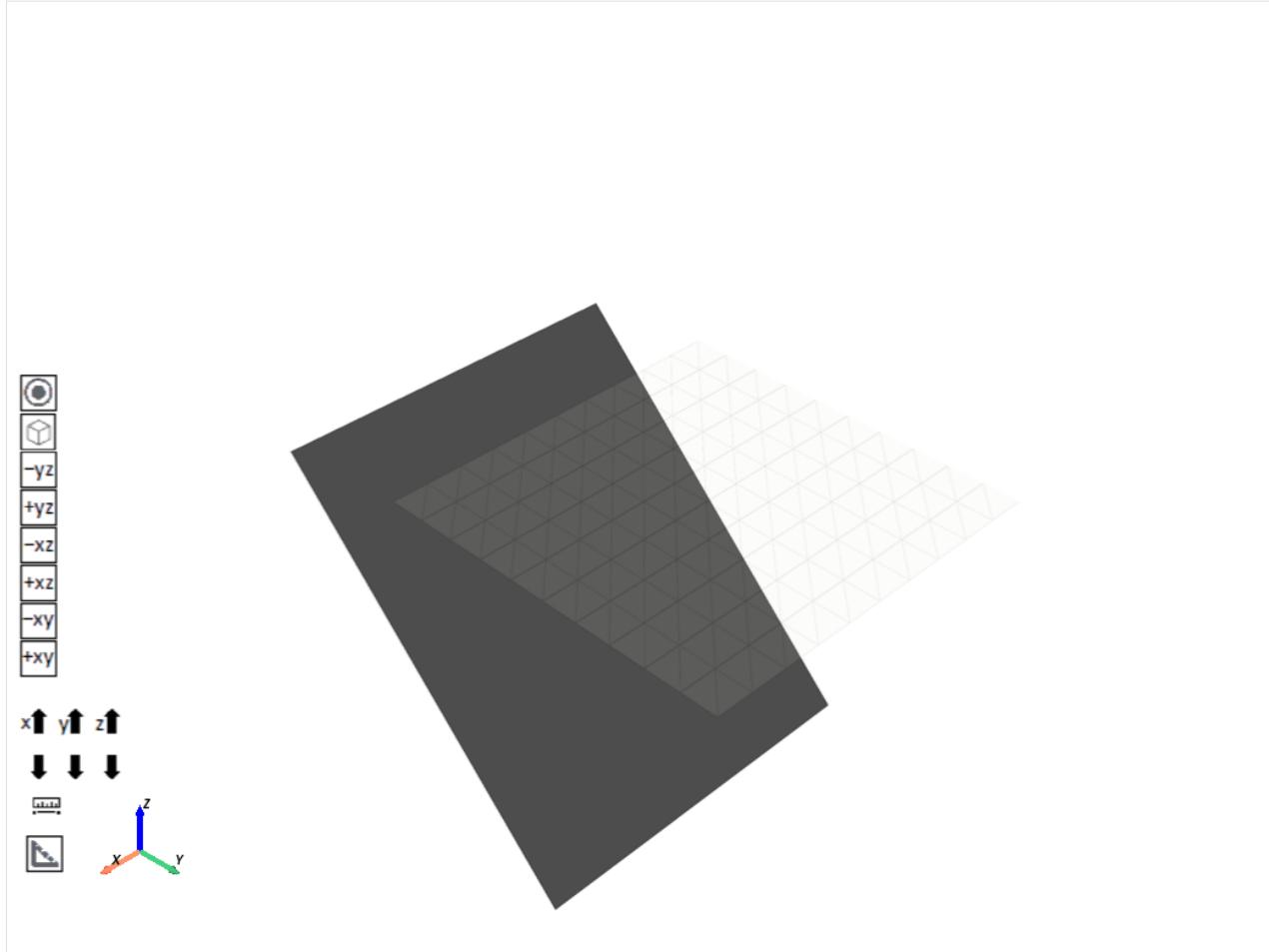
```
[8]: sketch.slot(Point2D([2, 0]), 4, 3, tag="Slot")
sketch.select("Slot")
sketch.plot_selection()
```



## Draw a box

You draw a box by specifying the center, width, and height.

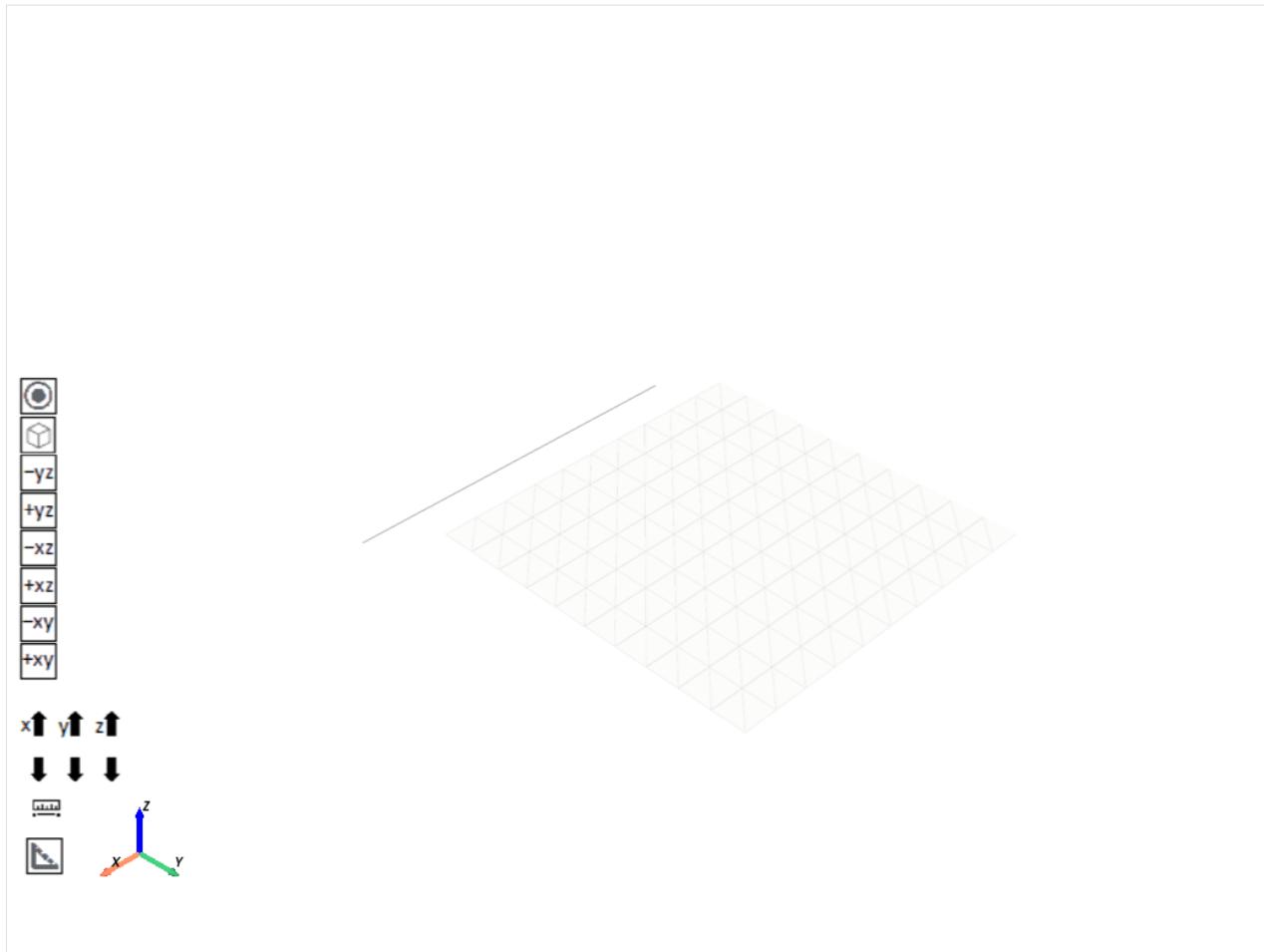
```
[9]: sketch.box(Point2D([2, 0]), 4, 5, tag="Box")
sketch.select("Box")
sketch.plot_selection()
```



## 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()
```

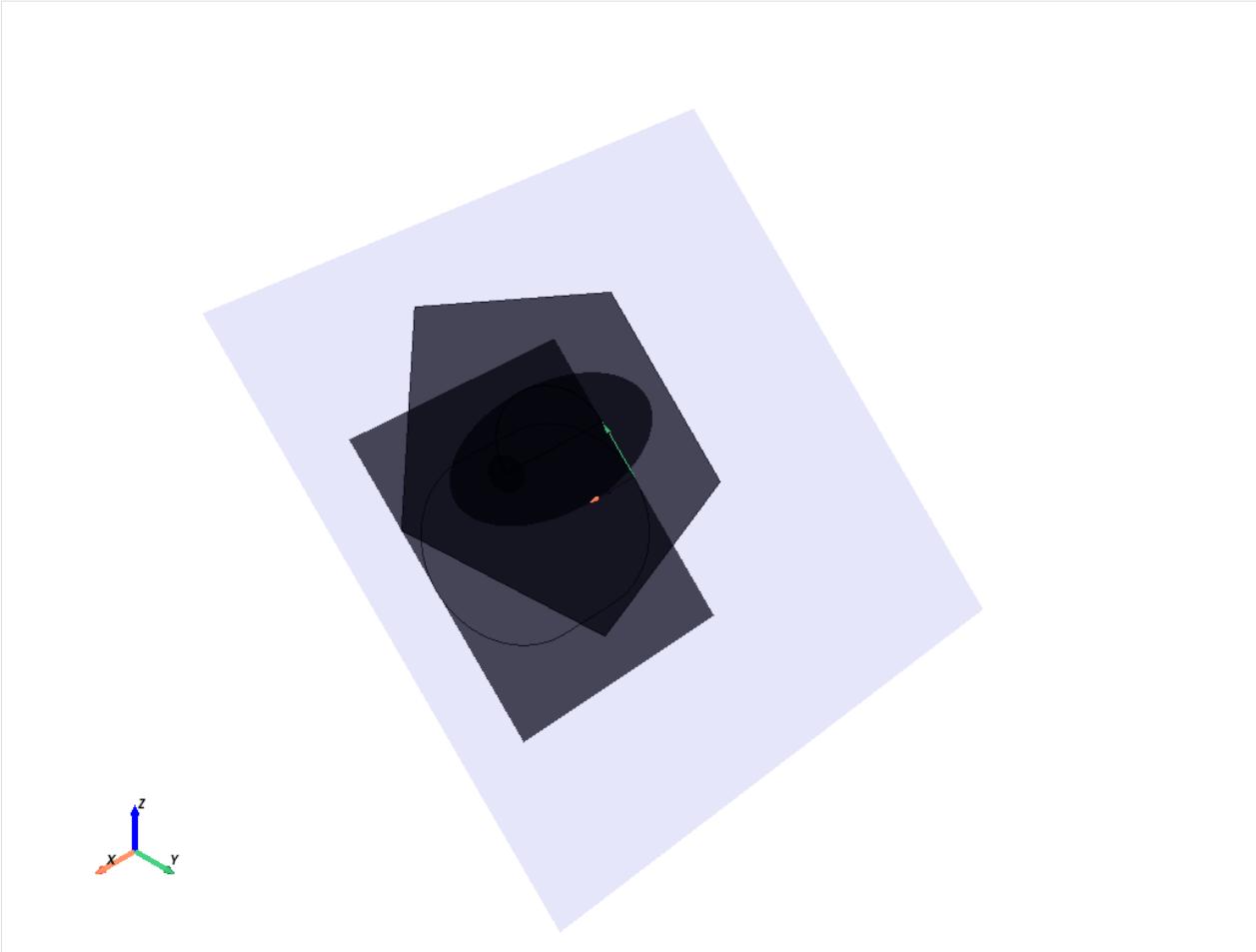


## Plot the sketch

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")
```



---

#### 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.

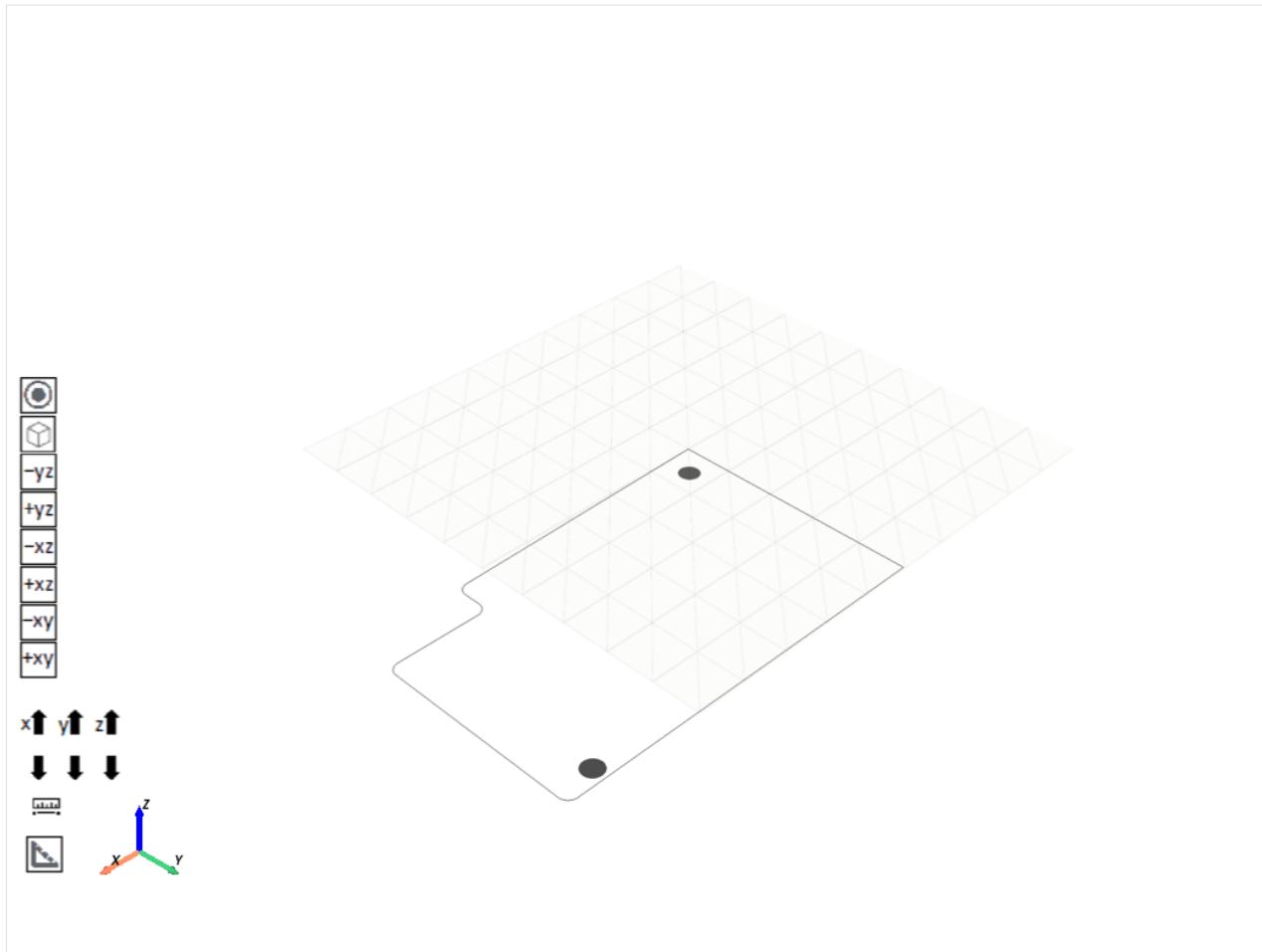
To define the sketch profile for the PCB, you create a sketch outline of individual Segment and Arc objects with two circular through-hole attachment points added within the profile boundary to maintain a single, closed sketch face.

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()
```



### Extrude multiple bodies

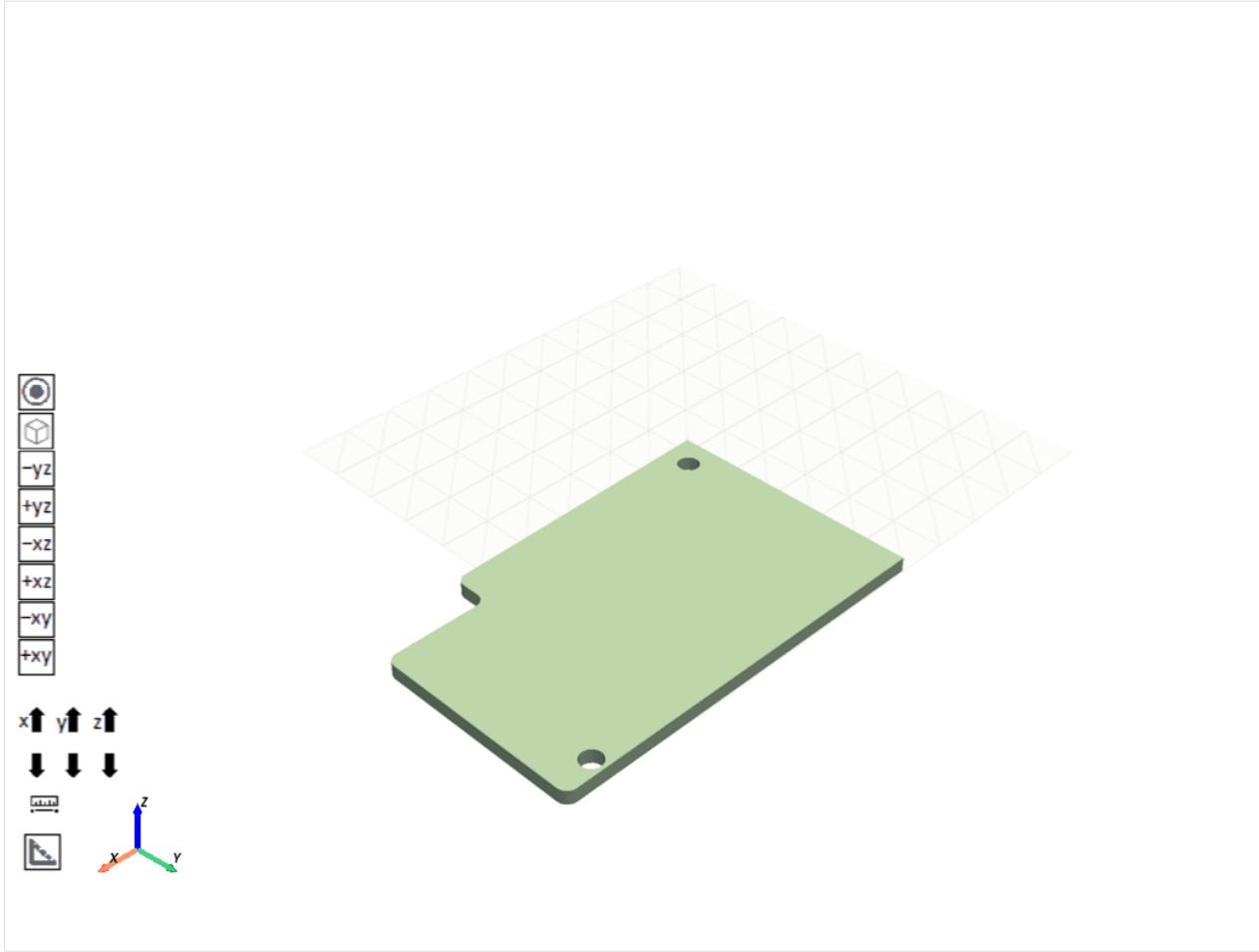
Establish a server connection and use the single sketch profile to extrude the board profile at multiple Z-offsets. Create a named selection from the resulting list of layer bodies.

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]: 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()
```



---

**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]: `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

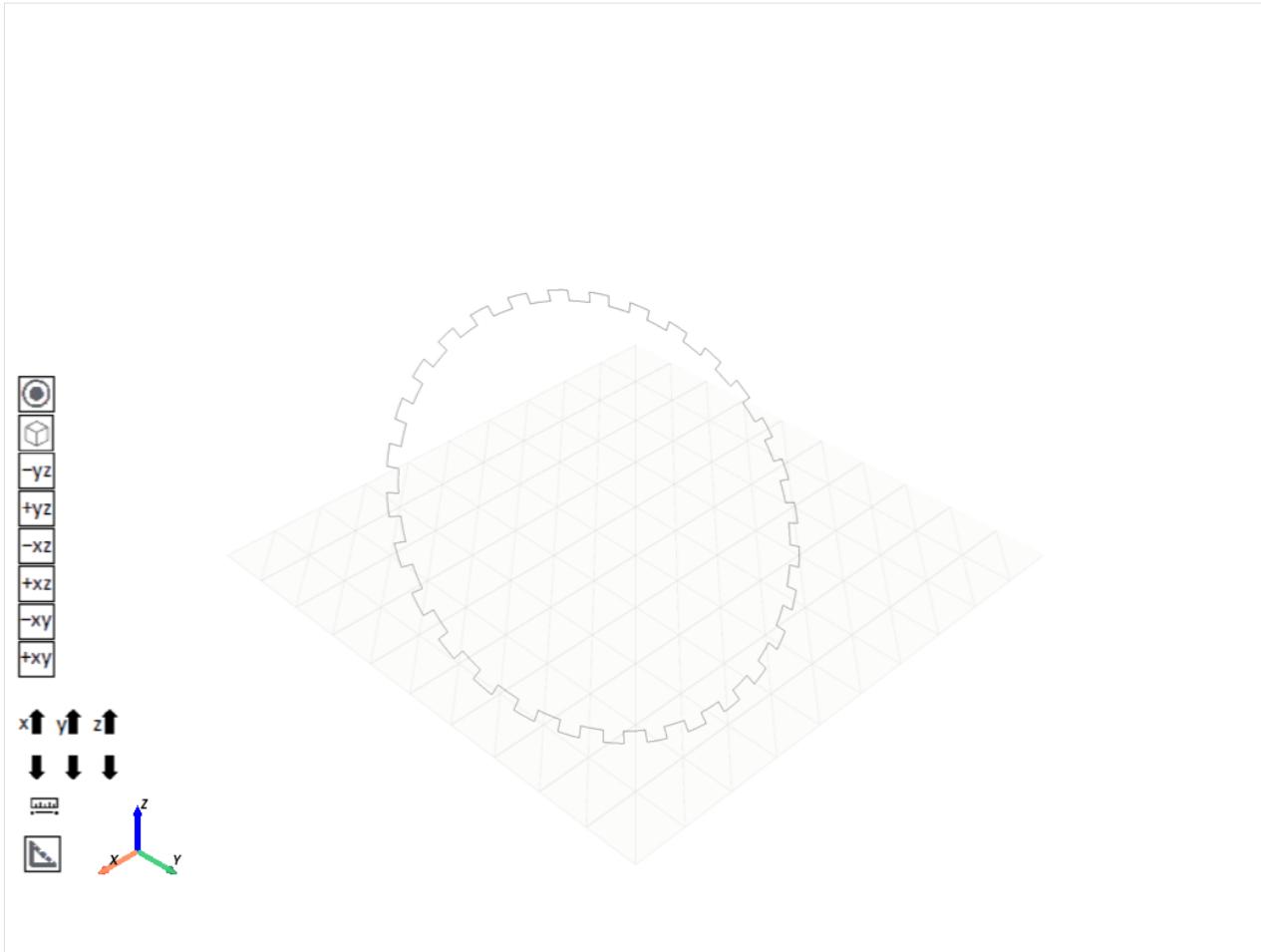
DummyGear sketches are simple gears that have straight teeth. While they do not ensure actual physical functionality, they might be useful for some simple playground tests.

Instantiate a new Sketch object and then define and plot a dummy gear.

[2]: `# 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()
```

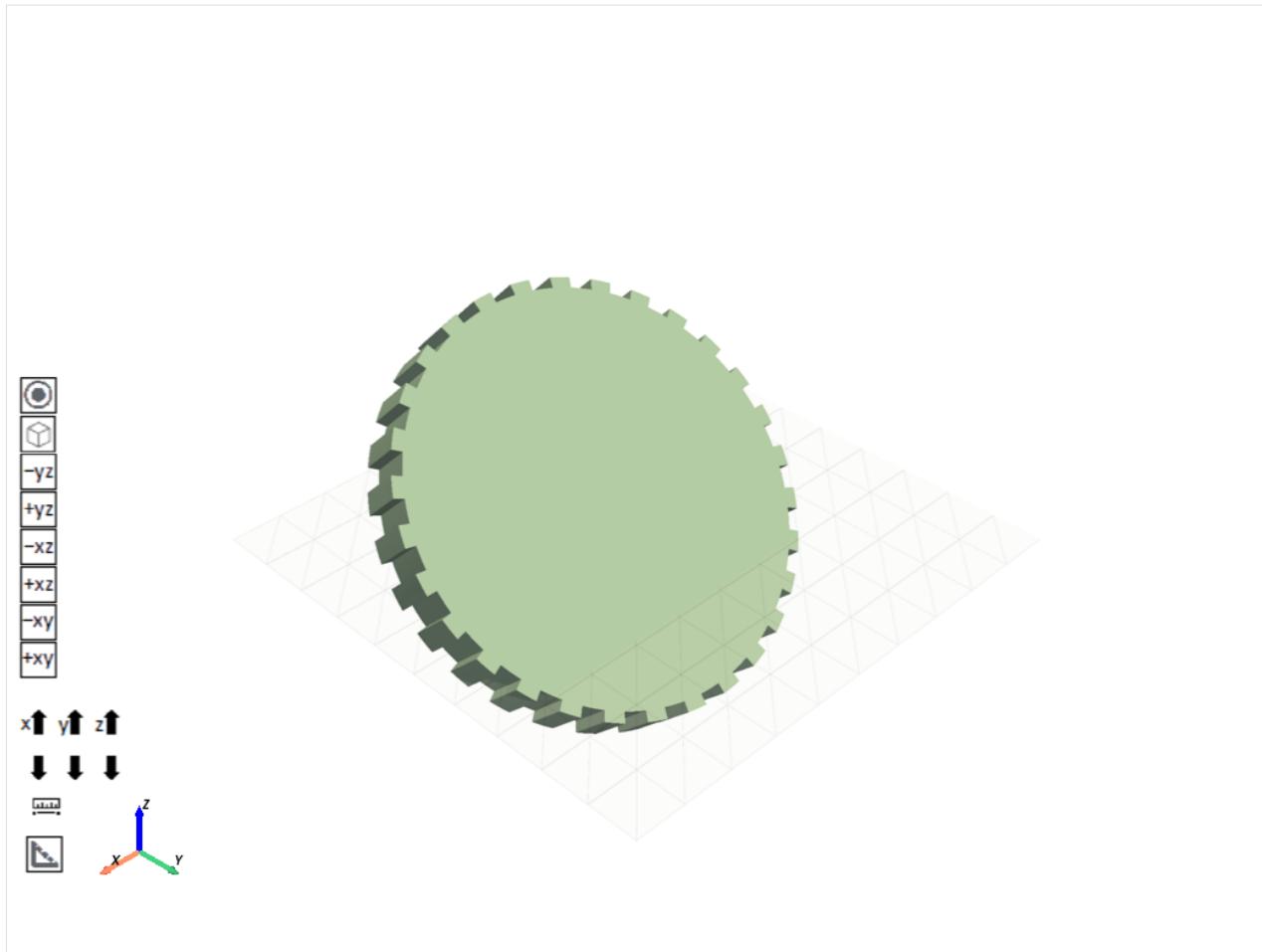


After creating the sketch, extrude it.

```
[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()
```



## 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)
```

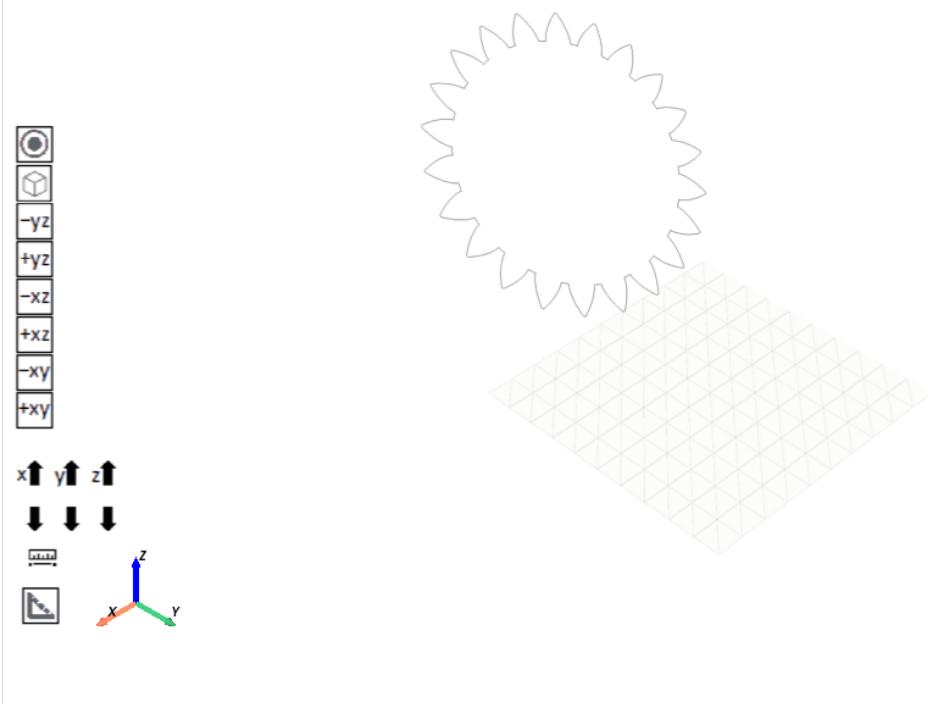
(continues on next page)

(continued from previous page)

```
n_teeth = 22

# Sketch spur gear
sketch.spur_gear(origin, module, pressure_angle, n_teeth)

# Plot spur gear
sketch.plot()
```

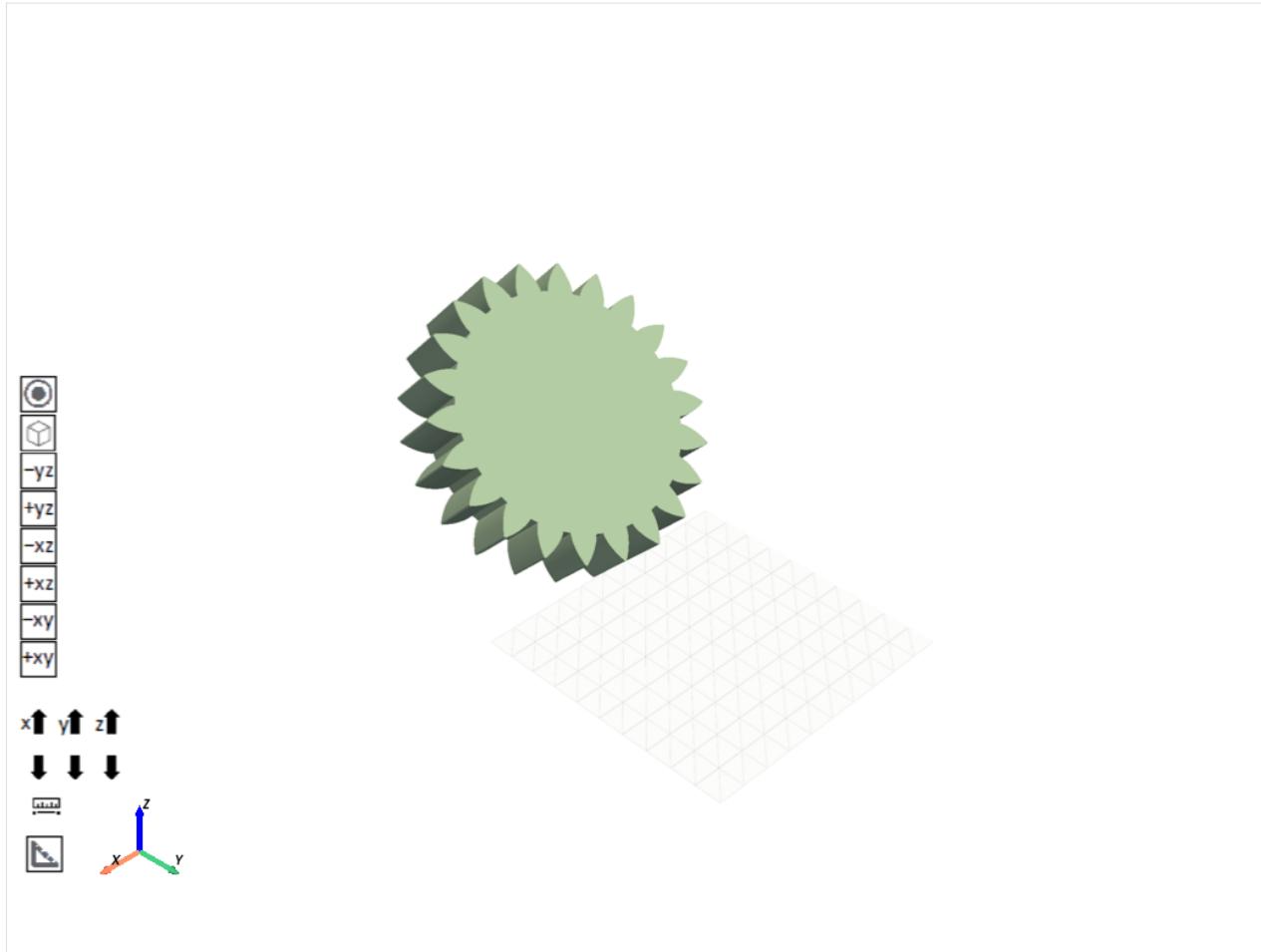


After creating the sketch, extrude it.

```
[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()
```



---

#### 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 0x1dd8c64edf0>
```

#### 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(
```

(continues on next page)

(continued from previous page)

```
"steel",
density,
[MaterialProperty(MaterialPropertyType.POISSON_RATIO, "PoissonRatio", poisson_
ratio)],
)
material.add_property(MaterialPropertyType.TENSILE_STRENGTH, "TensileProp", Quantity(45))
design.add_material(material)
```

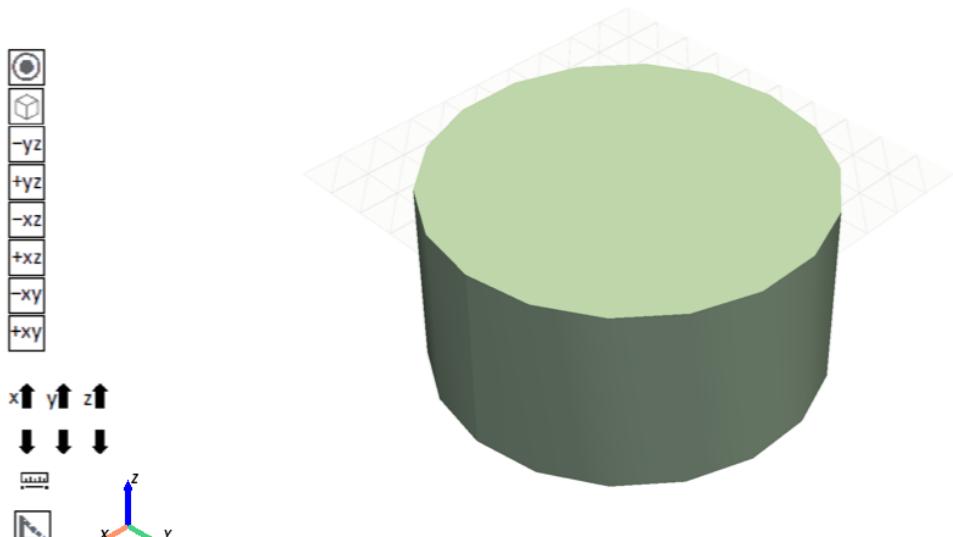
## Extrude sketch to create body

Extrude the sketch to create the body and then assign a material to it.

```
[5]: # 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()
```



---

**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 profile 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))
```

(continues on next page)

(continued from previous page)

```
.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 0x2285b2562b0>

### Extrude sketch to create design

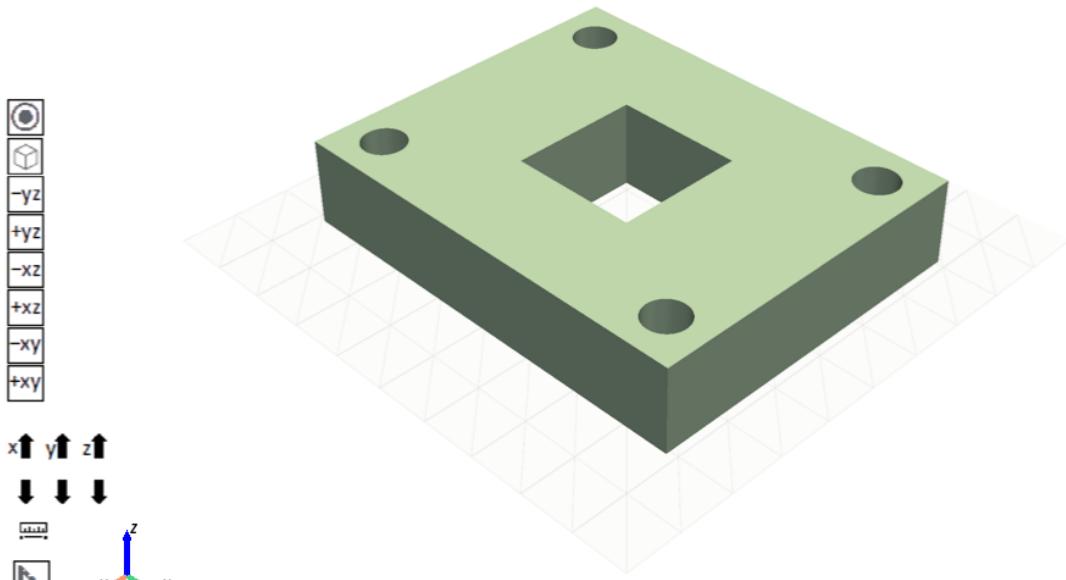
Establish a server connection and use the single sketch profile to extrude the base component at the Z axis. Create a named selection from the resulting list of bodies. In only three server calls, the design extrudes the four segments with the desired thickness.

[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()
```



## 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.

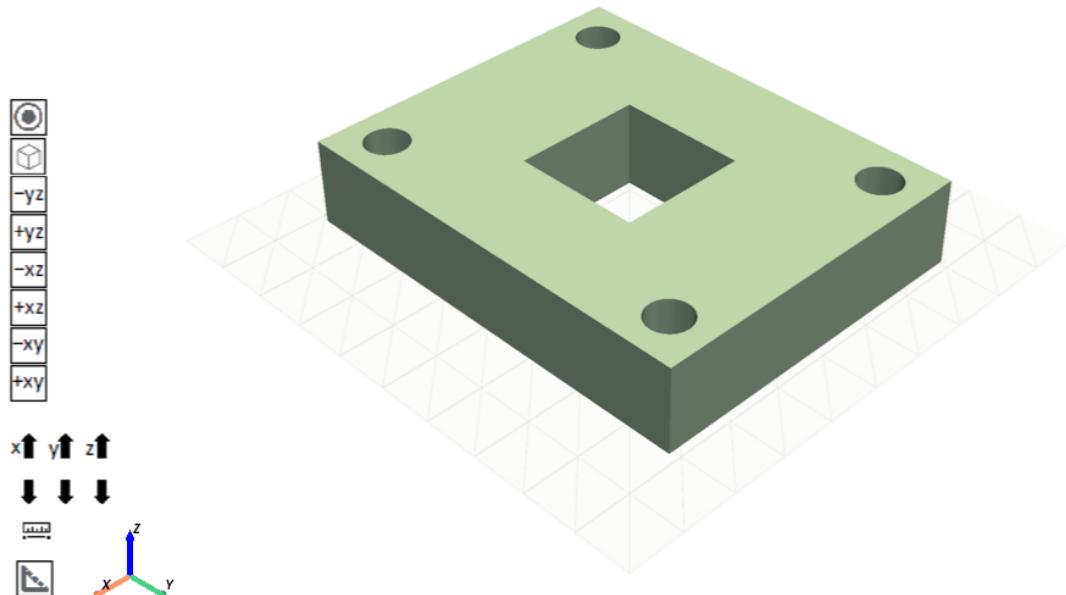
Create a sketch instance and then create a surface in the design with this sketch. For the sketch, it creates an ellipse, keeping the origin of the plane as its center.

```
[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)

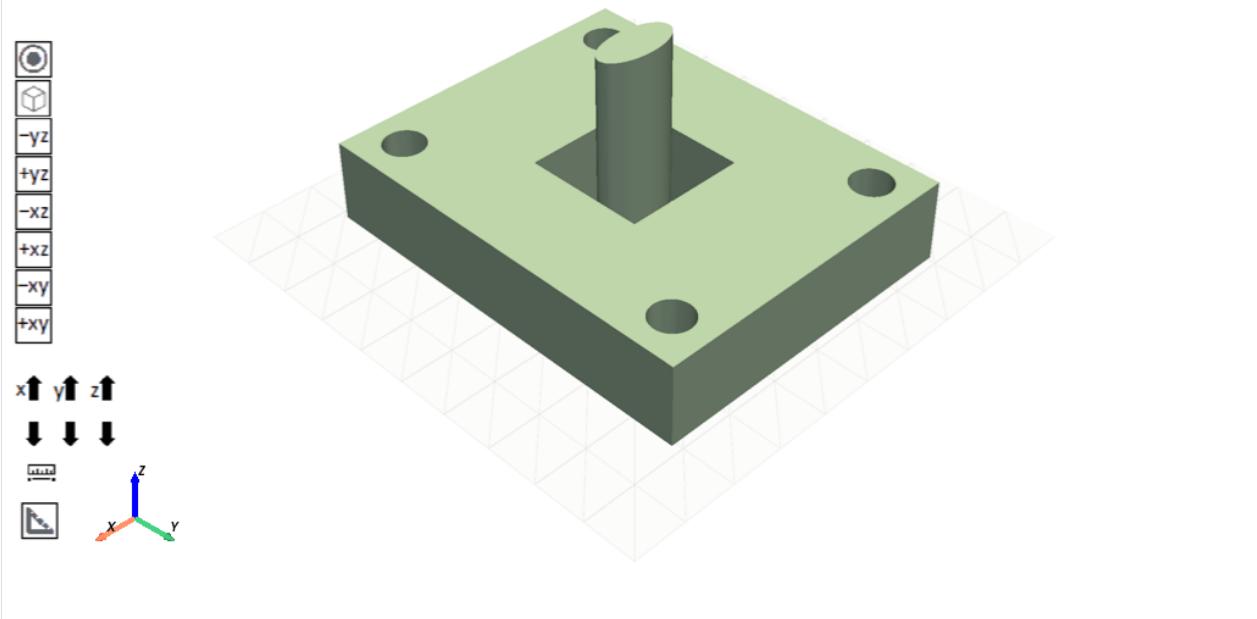
comp_str = repr(planar_component)
design.plot()
```



## 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.

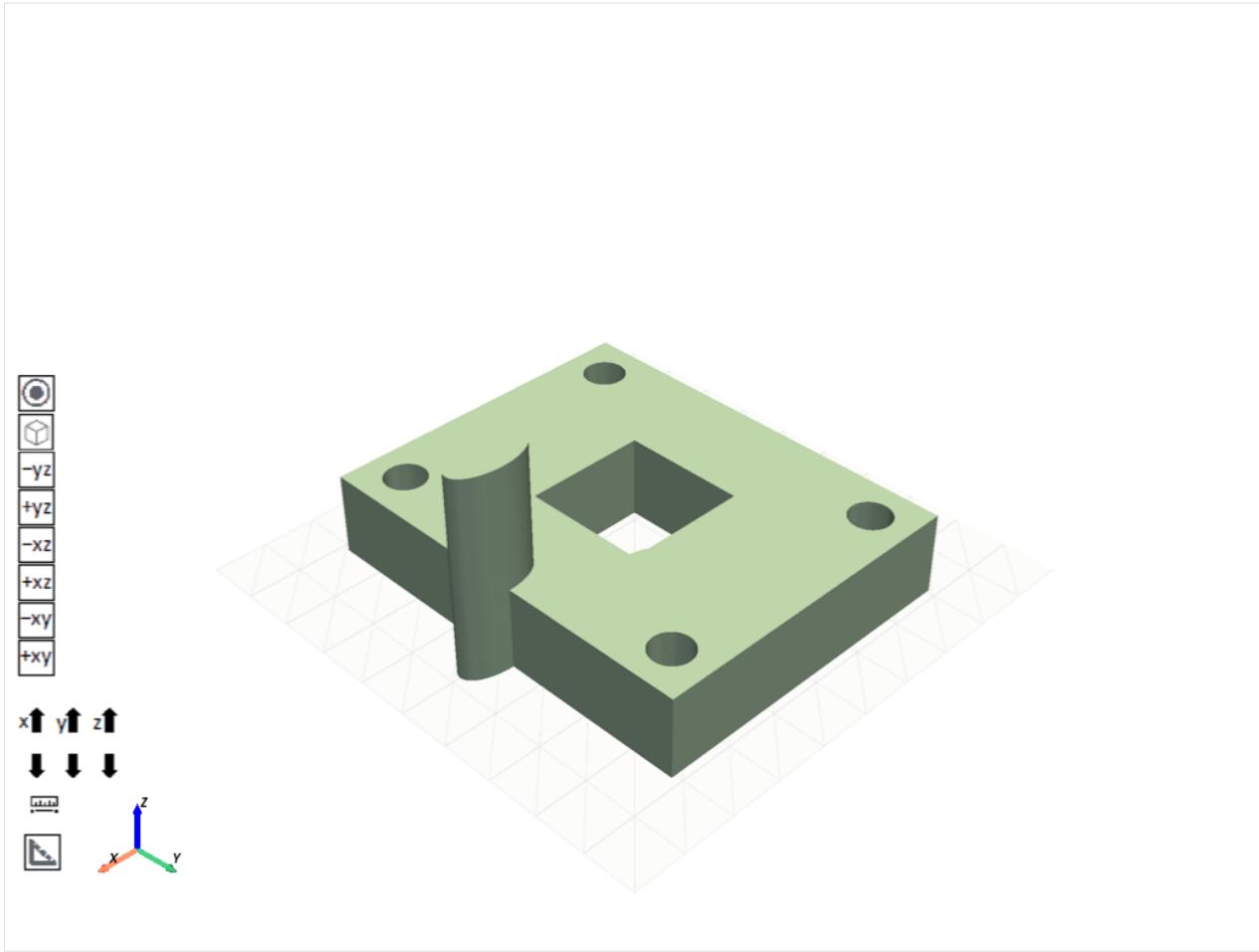
```
[5]: longer_body = design.extrude_face(
    "LongerEllipseFace", planar_body.faces[0], Quantity(5, UNITS.m)
)
design.plot()
```



## Translate body within plane

Use the `:func:translate()<ansys.geometry.core.designer.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()
```



---

**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 (0x233c5cb6580)
  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 (0x233c5cb6dc0)
  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

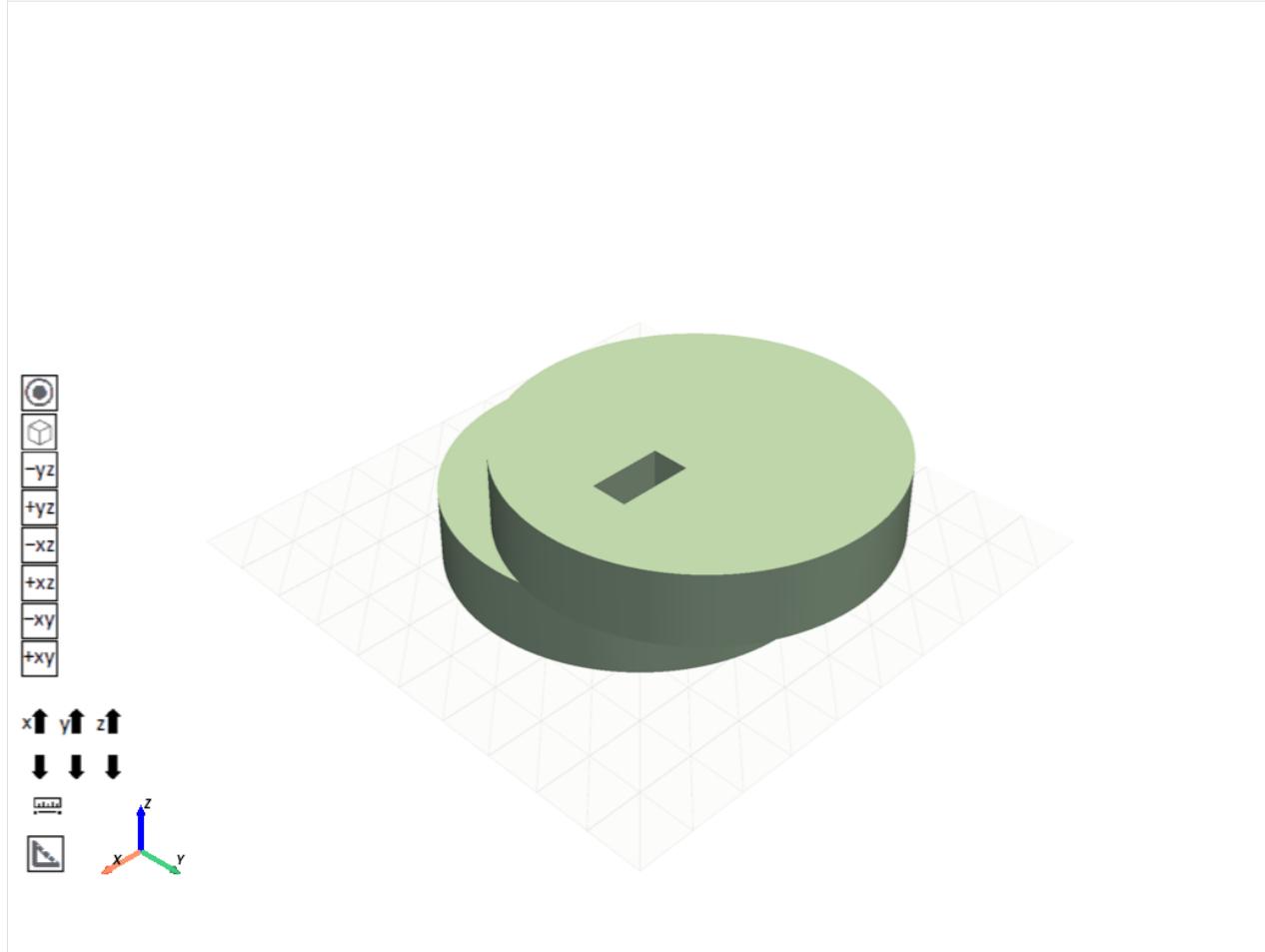
```
[5]: mesh = body.tessellate(merge=True)
mesh

[5]: PolyData (0x233c5cdc040)
  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()
```



---

#### 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")

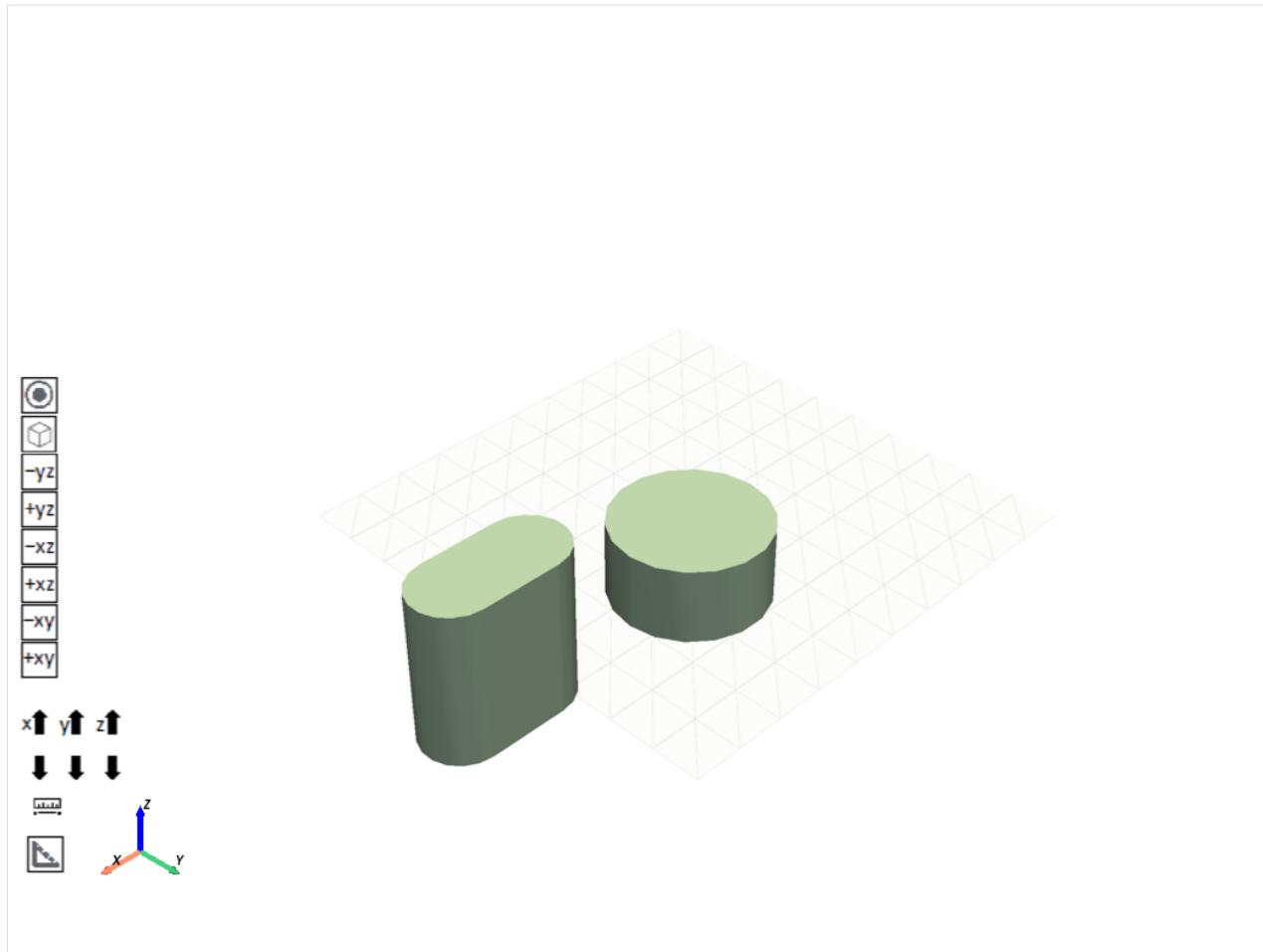
circle_sketch = Sketch()
circle_sketch.circle(Point2D([10, 10], UNITS.mm), Distance(10, UNITS.mm))

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()
```



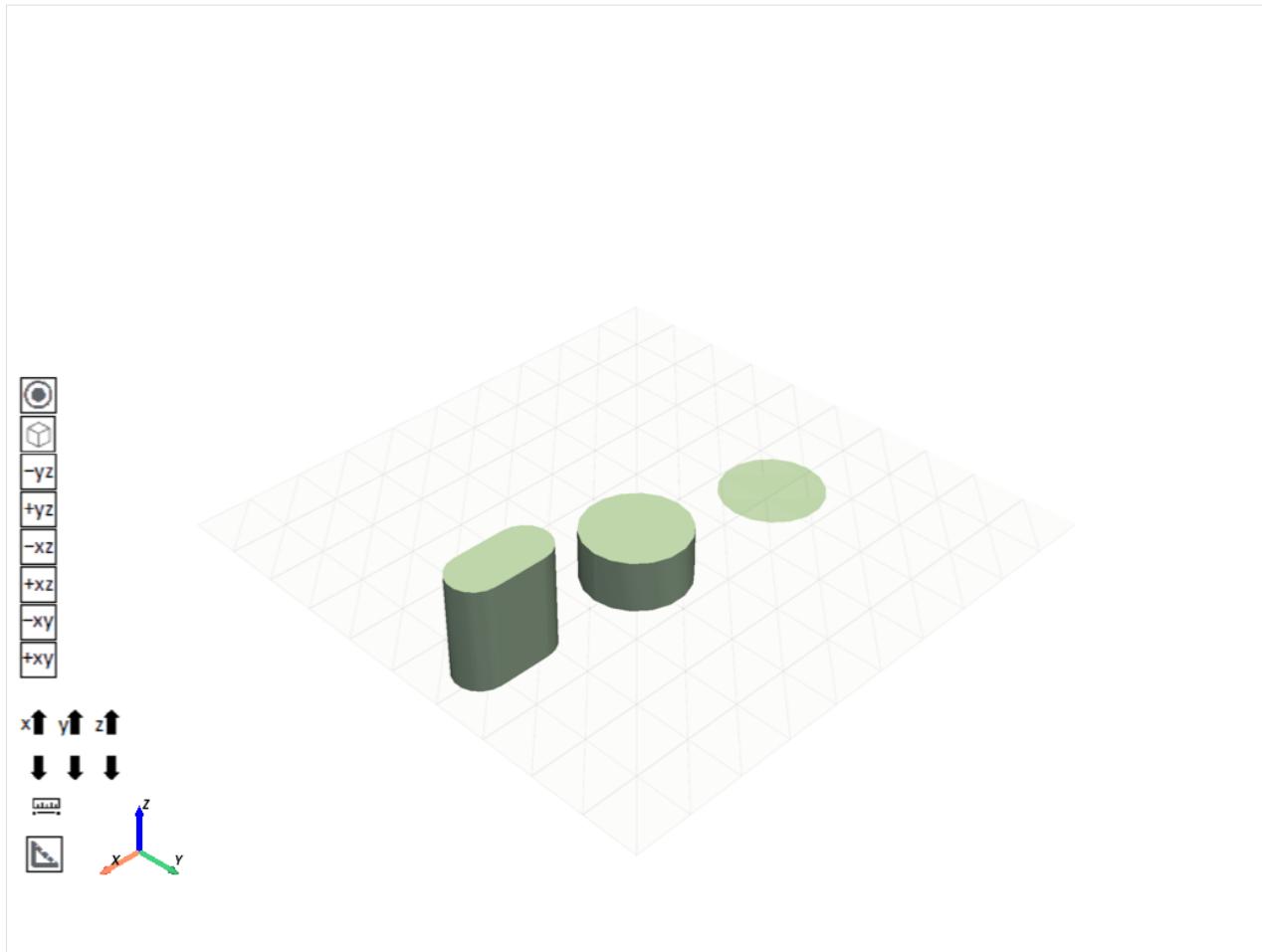
### Create nested component

Create a component that is nested under the previously created component and then create another cylinder from the previously used sketch.

```
[3]: 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()
```



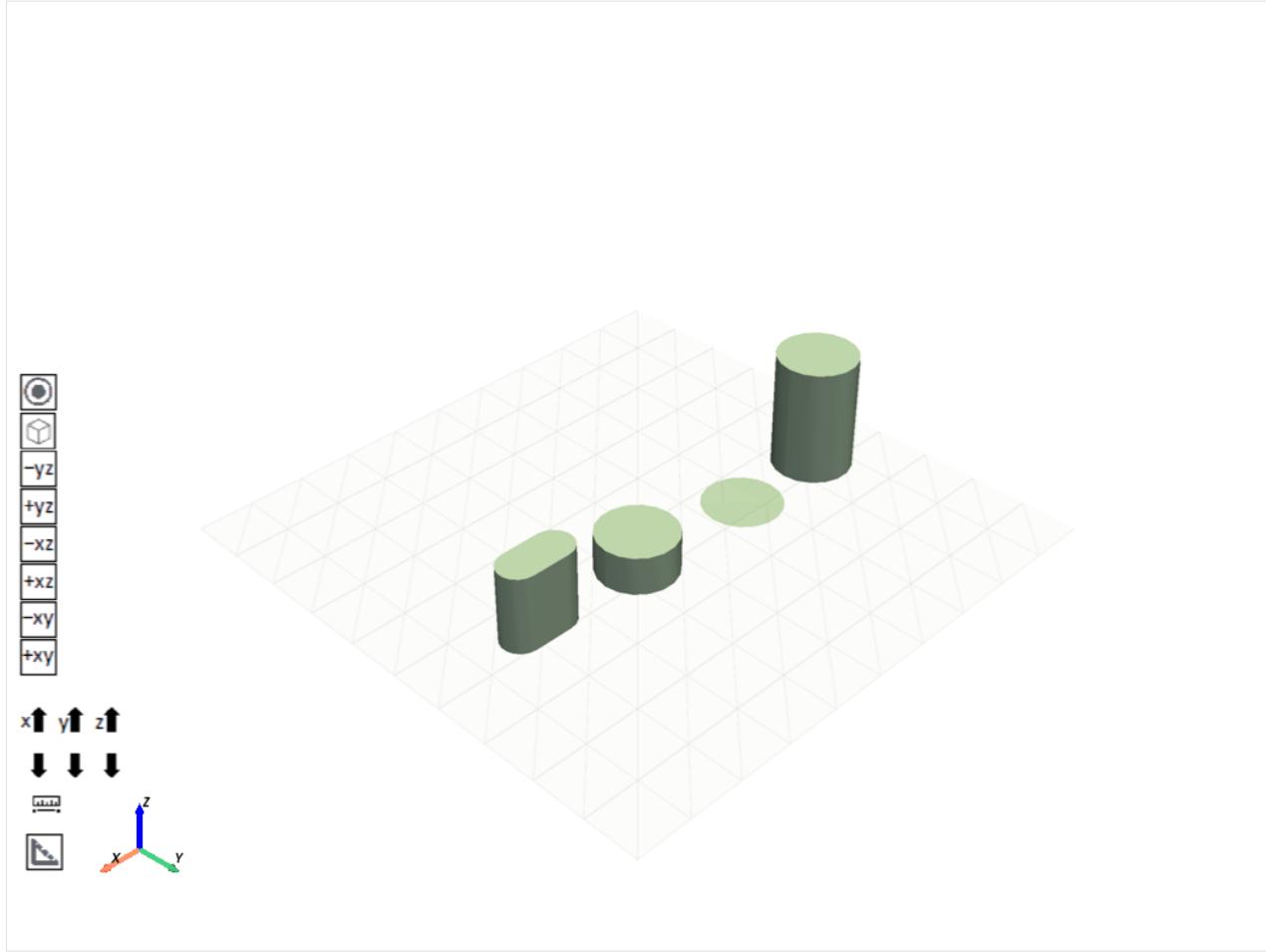
### Use surfaces from body to create additional bodies

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]: 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()
```



---

#### 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_docker_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_docker_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

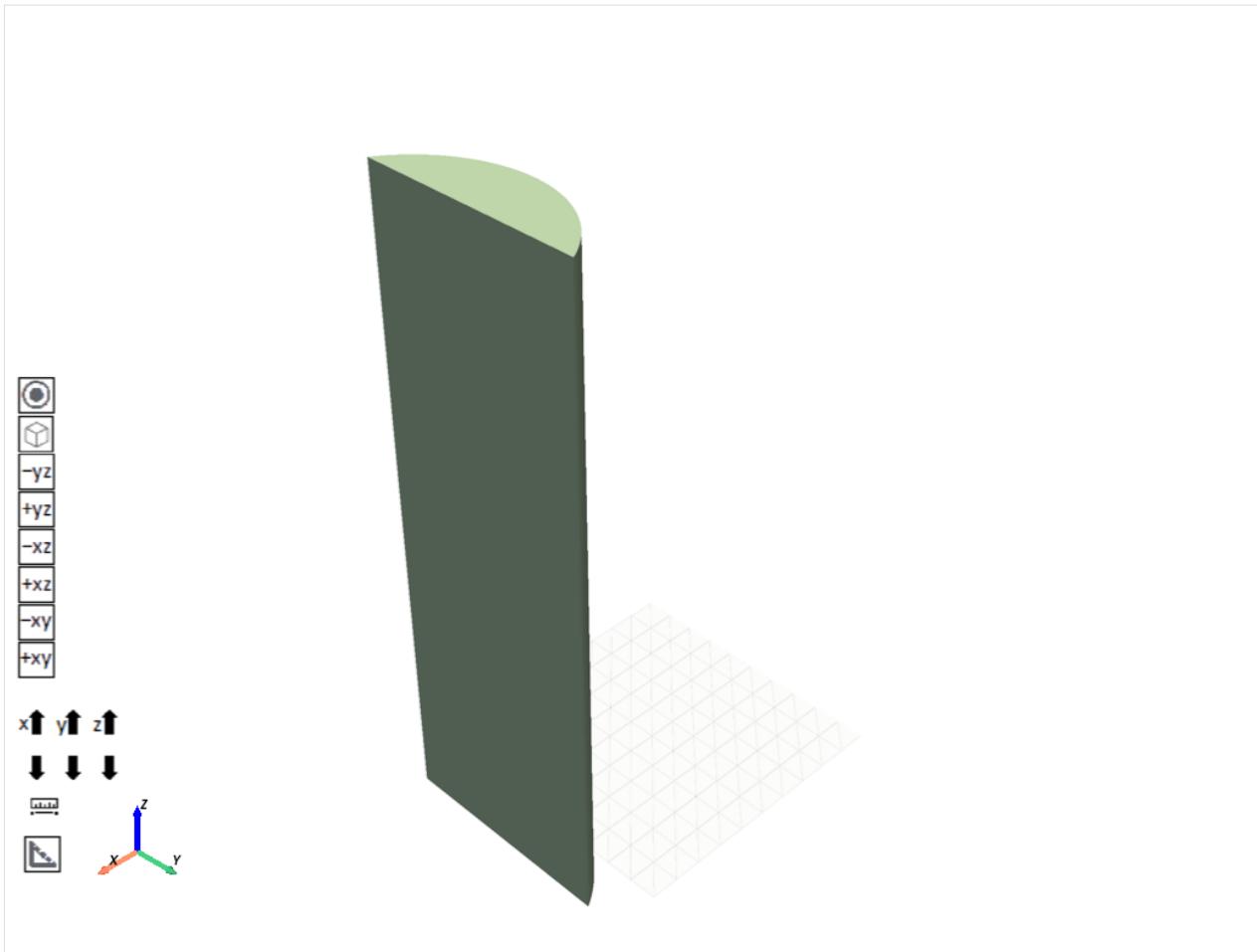
To perform an intersection operation on the bodies, first set up the bodies.

```
[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)
```



The final remaining body is the `prism` body because the `cylin` body has been consumed.

```
[8]: print(design.bodies)
[  
    ansys.geometry.core.designer.Body 0x19815894250  
        Name : Prism  
        Exists : True  
        Parent component : intersection_design  
        MasterBody : 0:22  
        Surface body : False
]
```

## Perform a union operation

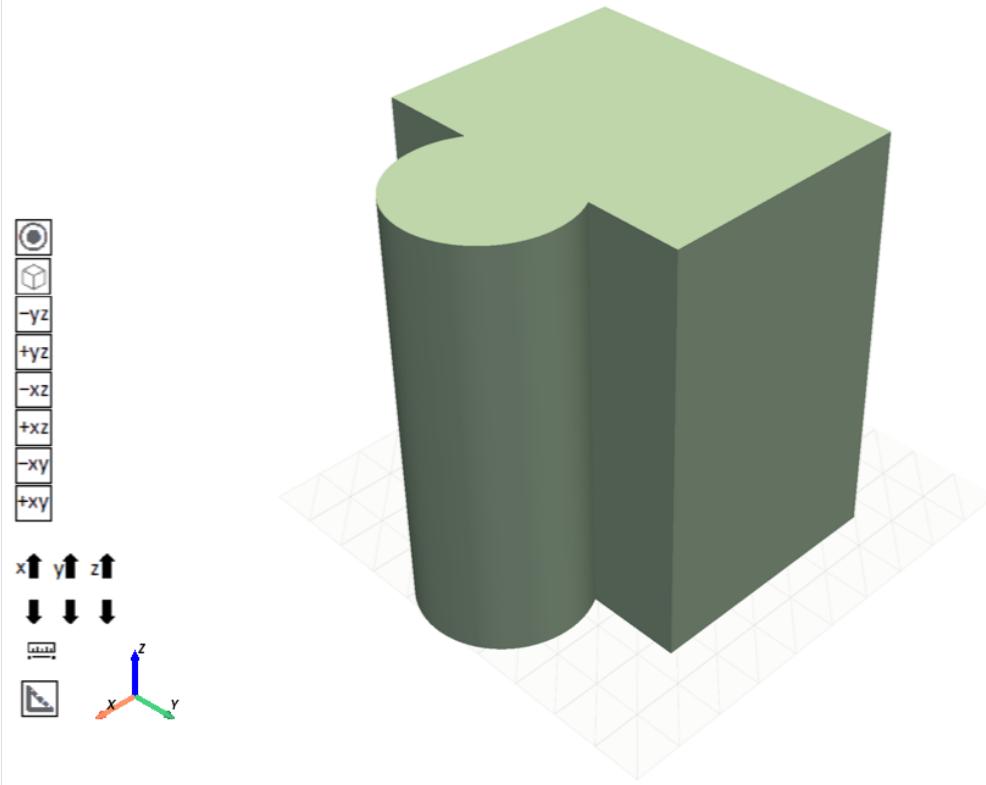
To carry out a union operation on the bodies, first set up the bodies.

```
[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)
```



The final remaining body is the `prism` body because the `cylin` body has been consumed.

```
[11]: print(design.bodies)

[
    ansys.geometry.core.designer.Body 0x19815909b20
        Name : Prism
        Exists : True
```

(continues on next page)

(continued from previous page)

```

Parent component      : union_design
MasterBody           : 0:22
Surface body         : False
]

```

## Perform a subtraction operation

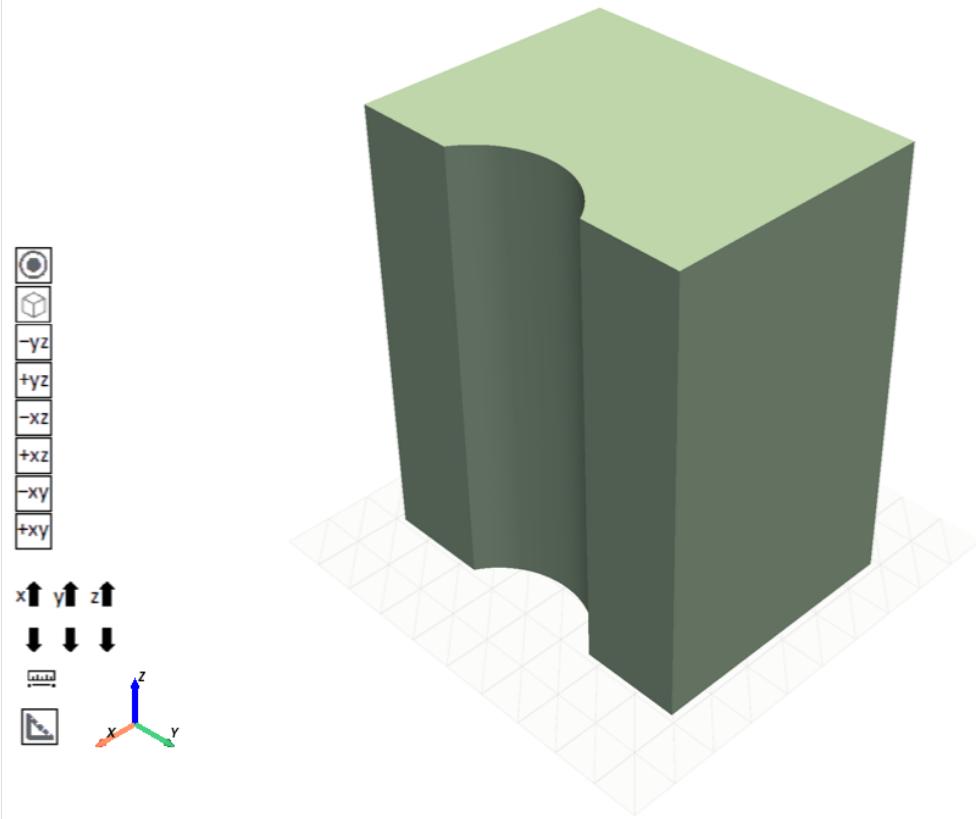
To perform a subtraction operation on the bodies, first set up the bodies.

```
[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)
```



The final remaining body is the `prism` body because the `cylin` body has been consumed.

```
[14]: print(design.bodies)
```

```
[
ansys.geometry.core.designer.Body 0x19815b89280
  Name          : Prism
  Exists        : True
  Parent component : subtraction_design
  MasterBody    : 0:22
  Surface body   : False
]
```

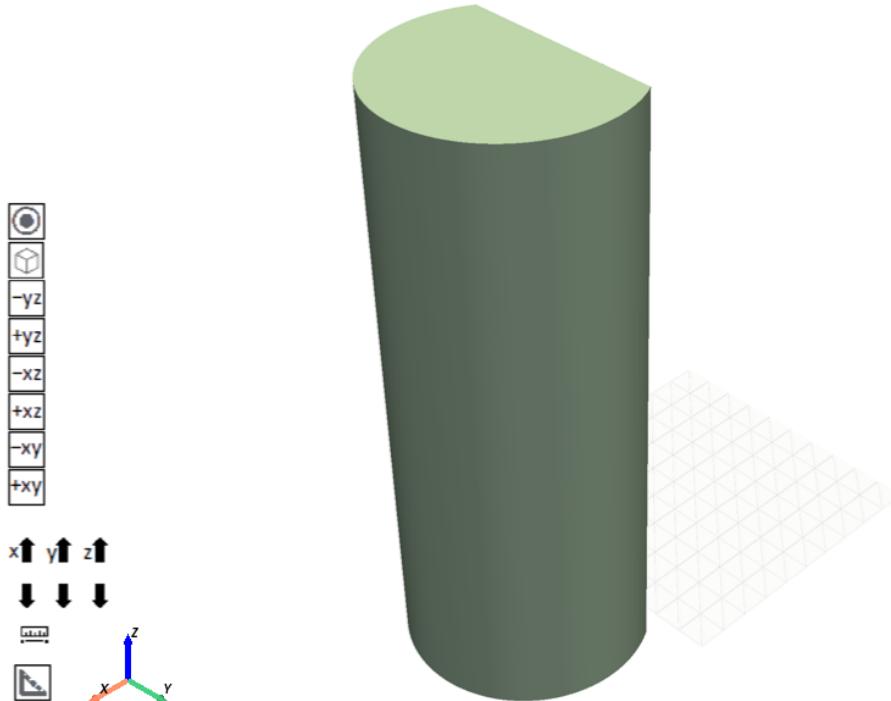
If you perform this action inverting the order of the bodies (that is, `cylin.subtract(prism)`), you can see the difference in the resulting shape of the body.

```
[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)
```



In this case, the final remaining body is the `cylin` body because the `prism` body has been consumed.

```
[16]: print(design.bodies)
[
    ansys.geometry.core.designer.Body 0x1981769ac40
        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](#).

---



## **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](mailto: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
```

## 5.5 Build the documentation

---

**Note:** To build the documentation, you must have the Geometry Service installed and running on your machine because it is used to generate the examples in the documentation. It is also recommended that the service is running as a Docker container.

If you do not have the Geometry Service installed, you can still build the documentation, but the examples are not generated. To build the documentation without the examples, define the following environment variable:

```
# On Linux or macOS
export BUILD_EXAMPLES=false

# On Windows CMD
set BUILD_EXAMPLES=false

# On Windows PowerShell
$env:BUILD_EXAMPLES="false"
```

To build the documentation locally, you must run this command to install the documentation dependencies:

```
pip install -e .[doc]
```

Then, navigate to the docs directory and run this command:

```
# On Linux or macOS
make html

# On Windows
./make.bat html
```

The documentation is built in the docs/\_build/html directory.

You can clean the documentation build by running this command:

```
# On Linux or macOS
make clean

# On Windows
./make.bat clean
```

## 5.6 Run tests

PyAnsys Geometry uses `pytest` for testing.

### 5.6.1 Prerequisites

Prior to running the tests, you must run this command to install the test dependencies:

```
pip install -e .[tests]
```

Make sure to define the port and host of the service using the following environment variables:

```
# On Linux or macOS
export ANSRV_GEO_PORT=5000
export ANSRV_GEO_HOST=localhost

# On Windows CMD
set ANSRV_GEO_PORT=5000
set ANSRV_GEO_HOST=localhost
```

(continues on next page)

(continued from previous page)

```
# On Windows PowerShell
$env:ANSRV_GEO_PORT=5000
$env:ANSRV_GEO_HOST="localhost"
```

## 5.6.2 Running the tests

To run the tests, navigate to the root directory of the repository and run this command:

```
pytest
```

---

**Note:** The tests require the Geometry Service to be installed and running on your machine. The tests fail if the service is not running. It is expected for the Geometry Service to be running as a Docker container.

If you do not have the Geometry Service running as a Docker container, but you have it running on your machine, you can still run the tests with the following argument:

```
pytest --use-existing-service=yes
```

---

**ASSETS**

In this section, users are able to download a set of assets related to PyAnsys Geometry.

## 6.1 Documentation

The following links 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 3.12. 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 by running `-d wheelhouse` (this is required for unzipping to a directory on Windows) 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
- Linux wheelhouse for Python 3.12

### **6.2.2 Windows**

- Windows wheelhouse for Python 3.9
- Windows wheelhouse for Python 3.10
- Windows wheelhouse for Python 3.11
- Windows wheelhouse for Python 3.12

### **6.2.3 MacOS**

- MacOS wheelhouse for Python 3.9
- MacOS wheelhouse for Python 3.10
- MacOS wheelhouse for Python 3.11
- MacOS wheelhouse for Python 3.12

## **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.

### **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

## PYTHON MODULE INDEX

### a

ansys.geometry.core, ??  
ansys.geometry.core.connection, ??  
ansys.geometry.core.connection.backend, ??  
ansys.geometry.core.connection.client, ??  
ansys.geometry.core.connection.conversions,  
    ??  
ansys.geometry.core.connection.defaults, ??  
ansys.geometry.core.connection.docker\_instance,  
    ??  
ansys.geometry.core.connection.launcher, ??  
ansys.geometry.core.connection.product\_instance,  
    ??  
ansys.geometry.core.connection.validate, ??  
ansys.geometry.core.designer, ??  
ansys.geometry.core.designer.beam, ??  
ansys.geometry.core.designer.body, ??  
ansys.geometry.core.designer.component, ??  
ansys.geometry.core.designer.coordinate\_system,  
    ??  
ansys.geometry.core.designer.design, ??  
ansys.geometry.core.designer.designpoint, ??  
ansys.geometry.core.designer.edge, ??  
ansys.geometry.core.designer.face, ??  
ansys.geometry.core.designer.part, ??  
ansys.geometry.core.designer.selection, ??  
ansys.geometry.core.errors, ??  
ansys.geometry.core.logger, ??  
ansys.geometry.core.materials, ??  
ansys.geometry.core.materials.material, ??  
ansys.geometry.core.materials.property, ??  
ansys.geometry.core.math, ??  
ansys.geometry.core.math.bbox, ??  
ansys.geometry.core.math.constants, ??  
ansys.geometry.core.math.frame, ??  
ansys.geometry.core.math.matrix, ??  
ansys.geometry.core.math.plane, ??  
ansys.geometry.core.math.point, ??  
ansys.geometry.core.math.vector, ??  
ansys.geometry.core.misc, ??  
ansys.geometry.core.misc.accuracy, ??  
ansys.geometry.core.misc.auxiliary, ??  
ansys.geometry.core.misc.checks, ??  
ansys.geometry.core.misc.measurements, ??  
ansys.geometry.core.misc.options, ??  
ansys.geometry.core.misc.units, ??  
ansys.geometry.core.modeler, ??  
ansys.geometry.core.plotting, ??  
ansys.geometry.core.plotting.plotter, ??  
ansys.geometry.core.plotting.plotter\_helper,  
    ??  
ansys.geometry.core.plotting.plotting\_types,  
    ??  
ansys.geometry.core.plotting.trame\_gui, ??  
ansys.geometry.core.plotting.widgets, ??  
ansys.geometry.core.plotting.widgets.button,  
    ??  
ansys.geometry.core.plotting.widgets.displace\_arrows,  
    ??  
ansys.geometry.core.plotting.widgets.measure,  
    ??  
ansys.geometry.core.plotting.widgets.ruler,  
    ??  
ansys.geometry.core.plotting.widgets.show\_design\_point,  
    ??  
ansys.geometry.core.plotting.widgets.view\_button,  
    ??  
ansys.geometry.core.plotting.widgets.widget,  
    ??  
ansys.geometry.core.primitives, ??  
ansys.geometry.core.primitives.circle, ??  
ansys.geometry.core.primitives.cone, ??  
ansys.geometry.core.primitives.curve\_evaluation,  
    ??  
ansys.geometry.core.primitives.cylinder, ??  
ansys.geometry.core.primitives.ellipse, ??  
ansys.geometry.core.primitives.line, ??  
ansys.geometry.core.primitives.parameterization,  
    ??  
ansys.geometry.core.primitives.sphere, ??  
ansys.geometry.core.primitives.surface\_evaluation,  
    ??  
ansys.geometry.core.primitives.torus, ??  
ansys.geometry.core.sketch, ??

```
ansys.geometry.core.sketch.arc, ??  
ansys.geometry.core.sketch.box, ??  
ansys.geometry.core.sketch.circle, ??  
ansys.geometry.core.sketch.edge, ??  
ansys.geometry.core.sketch.ellipse, ??  
ansys.geometry.core.sketch.face, ??  
ansys.geometry.core.sketch.gears, ??  
ansys.geometry.core.sketch.polygon, ??  
ansys.geometry.core.sketch.segment, ??  
ansys.geometry.core.sketch.sketch, ??  
ansys.geometry.core.sketch.slot, ??  
ansys.geometry.core.sketch.trapezoid, ??  
ansys.geometry.core.sketch.triangle, ??  
ansys.geometry.core.tools, ??  
ansys.geometry.core.tools.measurement_tools,  
    ??  
ansys.geometry.core.tools.problem_areas, ??  
ansys.geometry.core.tools.repair_tool_message,  
    ??  
ansys.geometry.core.tools.repair_tools, ??  
ansys.geometry.core.typing, ??
```