topologicpy

Release 0.7.77

Wassim Jabi

Nov 24, 2024

CONTENTS:

CHAPTER

ONE

AN AI-POWERED SPATIAL MODELLING AND ANALYSIS SOFTWARE LIBRARY FOR ARCHITECTURE, ENGINEERING, AND CONSTRUCTION

1.1 Introduction

Welcome to topologicpy (rhymes with apple pie). topologicpy is an open-source python 3 implementation of Topologic which is a powerful spatial modelling and analysis software library that revolutionizes the way you design architectural spaces, buildings, and artefacts. Topologic's advanced features enable you to create hierarchical and topological information-rich 3D representations that offer unprecedented flexibility and control in your design process. With the integration of geometry, topology, information, and artificial intelligence, Topologic enriches Building Information Models with Building *Intelligence* Models.

Two of Topologic's main strengths are its support for *defeaturing* and *encoded meshing*. By simplifying the geometry of a model and removing small or unnecessary details not needed for analysis, defeaturing allows for faster and more accurate analysis while maintaining topological consistency. This feature enables you to transform low-quality, heavy BIM models into high-quality, lightweight representations ready for rigorous analysis effortlessly. Encoded meshing allows you to use the same base elements available in your commercial BIM platform to cleanly build 3D information-encoded models that match your exacting specifications.

Topologic's versatility extends to entities with mixed dimensionalities, enabling structural models, for example, to be represented coherently. Lines can represent columns and beams, surfaces can represent walls and slabs, and volumes can represent solids. Even non-building entities like structural loads can be efficiently attached to the structure. This approach creates mixed-dimensional models that are highly compatible with structural analysis simulation software.

Topologic's graph-based representation makes it a natural fit for integrating with Graph Machine Learning (GML), an exciting new branch of artificial intelligence. With GML, you can process vast amounts of connected data and extract valuable insights quickly and accurately. Topologic's intelligent algorithms for graph and node classification take GML to the next level by using the extracted data to classify building typologies, predict associations, and complete missing information in building information models. This integration empowers you to leverage the historical knowledge embedded in your databases and make informed decisions about your current design projects. With Topologic and GML, you can streamline your workflow, enhance your productivity, and achieve your project goals with greater efficiency and precision.

Experience Topologic's comprehensive and well-documented Application Protocol Interface (API) and enjoy the freedom and flexibility that Topologic offers in your architectural design process. Topologic uses cutting-edge C++-based non-manifold topology (NMT) core technology (Open CASCADE), and python bindings. Interacting with Topologic is easily accomplished through a command-Line interface and scripts, visual data flow programming (VDFP) plugins for popular BIM software, and cloud-based interfaces through Streamlit. You can easily interact with Topologic in various ways to perform design and analysis tasks or even seamlessly customize and embed it in your own in-house software and workflows. Plus, Topologic includes several industry-standard methods for data transport including IFC, OBJ, BREP, HBJSON, CSV, as well serializing through cloud-based services such as Speckle.

Topologic's open-source philosophy and licensing (AGPLv3) enables you to achieve your design vision with minimal incremental costs, ensuring a high return on investment. You control and own your information outright, and nothing is ever trapped in an expensive subscription model. Topologic empowers you to build and share data apps with ease, giving you the flexibility to choose between local or cloud-based options and the peace of mind to focus on what matters most.

Join the revolution in architectural design with Topologic. Try it today and see the difference for yourself.

1.2 Installation

topologicpy can be installed using the **pip** command as such:

```
pip install topologicpy --upgrade
```

1.3 Prerequisites

topologicpy depends on the following python libraries which will be installed automatically from pip:

1.4 How to start using Topologic

1. Open your favourite python editor (jupyter notebook is highly recommended)
2. Type 'import topologicpy'
3. Start using the API

1.5 API Documentation

API documentation can be found at https://topologicpy.readthedocs.io

1.6 How to cite topologicpy

If you wish to cite the actual software, you can use:

Jabi, W. (2024). topologicpy. pypi.org. http://doi.org/10.5281/zenodo.11555172

To cite one of the main papers that defines topologicpy, you can use:

Jabi, W., & Chatzivasileiadi, A. (2021). Topologic: Exploring Spatial Reasoning Through Geometry, Topology, and Semantics. In S. Eloy, D. Leite Viana, F. Morais, & J. Vieira Vaz (Eds.), Formal Methods in Architecture (pp. 277–285). Springer International Publishing. https://doi.org/10.1007/978-3-030-57509-0_25

Or you can import the following .bib formatted references into your favourite reference manager

```
@misc{Jabi2024,
   author = {Wassim Jabi},
   doi = {https://doi.org/10.5281/zenodo.11555173},
   title = {topologicpy},
   url = {http://pypi.org/projects/topologicpy},
```

(continues on next page)

(continued from previous page)

```
   year = {2024},
}
```

```
  @inbook{Jabi2021,
   abstract = {Topologic is a software modelling library that supports a comprehensive␣
↪conceptual framework for the hierarchical spatial representation of buildings based on␣
↪the data structures and concepts of non-manifold topology (NMT). Topologic supports␣
↪conceptual design and spatial reasoning through the integration of geometry, topology,␣
↪and semantics. This enables architects and designers to reflect on their design␣
↪decisions before the complexities of building information modelling (BIM) set in. We␣
↪summarize below related work on NMT starting in the late 1980s, describe Topologic's␣
↪software architecture, methods, and classes, and discuss how Topologic's features␣
↪support conceptual design and spatial reasoning. We also report on a software␣
↪usability workshop that was conducted to validate a software evaluation methodology␣
↪and reports on the collected qualitative data. A reflection on Topologic's features␣
↪and software architecture illustrates how it enables a fundamental shift from pursuing␣
↪fidelity of design form to pursuing fidelity of design intent.},
   author = {Wassim Jabi and Aikaterini Chatzivasileiadi},
   city = {Cham},
   doi = {10.1007/978-3-030-57509-0_25},
   editor = {Sara Eloy and David Leite Viana and Franklim Morais and Jorge Vieira Vaz},
   isbn = {978-3-030-57509-0},
   journal = {Formal Methods in Architecture},
   pages = {277-285},
   publisher = {Springer International Publishing},
   title = {Topologic: Exploring Spatial Reasoning Through Geometry, Topology, and␣
↪Semantics},
   url = {https://link.springer.com/10.1007/978-3-030-57509-0_25},
   year = {2021},
}
```


1.6.1 topologicpy

topologicpy package

Submodules

topologicpy.ANN module

class topologicpy.ANN.**ANN**

Bases: `object`

Methods

DatasetByCSVPath(path[, taskType, description])	Returns a dataset according to the input CSV file path.
DatasetBySampleName(name)	Returns a dataset from the scikit-learn dataset samples.
DatasetSampleNames()	Returns the names of the available sample datasets from sci-kit learn.
DatasetSplit(X, y[, testRatio, randomState])	Splits the input dataset according to the input ratios.
Figures(model[, width, height, template, ...])	Creates Plotly Figures from the model data.
Hyperparameters([title, taskType, ...])	Returns a Hyperparameters dictionary based on the input parameters.
HyperparametersBySampleName(name)	Returns the suggested initial hyperparameters to use for the dataset named in the name input parameter.
Initialize(hyperparameters, dataset)	Initializes an ANN model with the input dataset and hyperparameters.
Load(model, path)	Loads the model state dictionary found at the input file path.
Metrics(model)	Returns the model performance metrics given the input labels and predictions, and the model's task type.
ModelData(model)	Returns the data of the model
Save(model, path[, overwrite])	Saves the model.
Test(model, hyperparameters, dataset)	Returns the labels (actual values) and predictions (predicted values) given the input model, features (X), and target (y).
Train(hyperparameters, dataset)	Trains the input model given the input features (X), and target (y).

static DatasetByCSVPath(*path*, *taskType='classification'*, *description=''*)

Returns a dataset according to the input CSV file path.

Parameters

path
[str] The path to the folder containing the necessary CSV and YML files.

taskType
[str , optional] The type of evaluation task. This can be 'classification' or 'regression'. The default is 'classification'.

description
[str , optional] The description of the dataset. In keeping with the scikit BUNCH class, this will be saved in the DESCR parameter.

Returns

sklearn.utils._bunch.Bunch
The created dataset.

static DatasetBySampleName(*name*)

Returns a dataset from the scikit-learn dataset samples.

Parameters

name
[str] The name of the dataset. This can be one of ['breast_cancer', 'california_housing', 'digits', 'iris', 'wine']

Returns

sklearn.utils._bunch.Bunch
The created dataset.

`static DatasetSampleNames()`

Returns the names of the available sample datasets from sci-kit learn.

Parameters

Returns

list
The list of names of available sample datasets

`static DatasetSplit(X, y, testRatio=0.3, randomState=42)`

Splits the input dataset according to the input ratios.

Parameters

X
[list] The list of features.

y
[list] The list of targets.

testRatio
[float , optional] The ratio of the dataset to reserve as unseen data for testing. The default is 0.3

randomState
[int , optional] The randomState parameter is used to ensure reproducibility of the results. When you set the randomState parameter to a specific integer value, it controls the shuffling of the data before splitting it into training and testing sets. This means that every time you run your code with the same randomState value and the same dataset, you will get the same split of the data. The default is 42 which is just a randomly picked integer number. Specify None for random sampling.

Returns

list
Returns the following list : [X_train, X_test, y_train,y_test] X_train is the list of features used for training X_test is the list of features used for testing y_train is the list of targets used for training y_test is the list of targets used for testing

`static Figures(model, width=900, height=600, template='plotly', colorScale='viridis', colorSamples=10)`

Creates Plotly Figures from the model data. For classification tasks this includes a confusion matrix, loss, and accuracy figures. For regression tasks this includes loss and MAE figures.

Parameters

model
[ANN Model] The input model.

width
[int , optional] The desired figure width in pixels. The default is 900.

height
[int , optional] The desired figure height in pixels. The default is 900.

template
[str , optional] The desired Plotly template to use for the scatter plot. This can be one of [‘ggplot2’, ‘seaborn’, ‘simple_white’, ‘plotly’, ‘plotly_white’, ‘plotly_dark’, ‘presentation’, ‘xgridoff’, ‘ygridoff’, ‘gridon’, ‘none’]. The default is “plotly”.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. “viridis”, “plasma”). The default is “viridis”. For a full list of names, see https://plotly.com/python/builtin-colorscales/.

colorSamples
[int , optional] The number of discrete color samples to use for displaying the data. The default is 10.

Returns

list
Returns a list of Plotly figures and a corresponding list of file names.

`static Hyperparameters`(*title='Untitled'*, *taskType='classification'*, *testRatio=0.3*, *validationRatio=0.2*, *hiddenLayers=[12, 12, 12]*, *learningRate=0.001*, *epochs=10*, *batchSize=1*, *patience=5*, *earlyStopping=True*, *randomState=42*, *crossValidationType='holdout'*, *kFolds=3*, *interval=1*, *mantissa=6*)

Returns a Hyperparameters dictionary based on the input parameters.

Parameters

title
[str , optional] The desired title for the dataset. The default is “Untitled”.

taskType
[str , optional] The desired task type. This can be either ‘classification’ or ‘regression’ (case insensitive). Classification is a type of supervised learning where the model is trained to predict categorical labels (classes) from input data. Regression is a type of supervised learning where the model is trained to predict continuous numerical values from input data.

testRatio
[float , optional] The split ratio between training and testing. The default is 0.3. This means that 70% of the data will be used for training/validation and 30% will be reserved for testing as unseen data.

validationRatio
[float , optional] The split ratio between training and validation. The default is 0.2. This means that 80% of the validation data (left over after reserving test data) will be used for training and 20% will be used for validation.

hiddenLayers
[list , optional] The number of hidden layers and the number of nodes in each layer. If you wish to have 3hidden layers with 8 nodes in the first 16 nodes in the second, and 4 nodes in the last layer, you specify [8,16,4]. The default is [12,12,12]

learningRate
[float, optional] The desired learning rate. The default is 0.001. See https://en.wikipedia.org/wiki/Learning_rate

epochs
[int , optional] The desired number of epochs. The default is 10. See https://en.wikipedia.org/wiki/Neural_network_(machine_learning)

batchSize
[int , optional] The desired number of samples that will be propagated through the network

at one time before the model's internal parameters are updated. Instead of updating the model parameters after every single training sample (stochastic gradient descent) or after the entire training dataset (batch gradient descent), mini-batch gradient descent updates the model parameters after a specified number of samples, which is determined by batchSize. The default is 1.

patience
[int , optional] The desired number of epochs with no improvement in the validation loss after which training will be stopped if early stopping is enabled.

earlyStopping
[bool , optional] If set to True, the training will stop if the validation loss does not improve after a certain number of epochs defined by patience. The default is True.

randomState
[int , optional] The randomState parameter is used to ensure reproducibility of the results. When you set the randomState parameter to a specific integer value, it controls the shuffling of the data before splitting it into training and testing sets. This means that every time you run your code with the same randomState value and the same dataset, you will get the same split of the data. The default is 42 which is just a randomly picked integer number. Specify None for random sampling.

crossValidationType
[str , optional] The desired type of cross-validation. This can be one of 'holdout' or 'k-fold'. The default is 'holdout'

kFolds
[int , optional] The number of splits (folds) to use if K-Fold cross validation is selected. The default is 5.

interval
[int , optional] The desired epoch interval at which to report and save metrics data. This must be less than the total number of epochs. The default is 1.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
Returns a dictionary with the following keys: 'title' 'task_type' 'test_ratio' 'validation_ratio' 'hidden_layers' 'learning_rate' 'epochs' 'batch_size' 'early_stopping' 'patience' 'random_state' 'cross_val_type' 'kFolds' 'interval' 'mantissa'

`static HyperparametersBySampleName`(*name*)

Returns the suggested initial hyperparameters to use for the dataset named in the name input parameter. You can get a list of available sample datasets using ANN.SampleDatasets().

Parameters

name
[str] The input name of the sample dataset. This must be one of ['breast_cancer', 'california_housing', 'digits', 'iris', 'wine']

Returns

dict
Returns a dictionary with the following keys: 'title' 'task_type' 'test_ratio' 'validation_ratio' 'hidden_layers' 'learning_rate' 'epochs' 'batch_size' 'early_stopping' 'patience' 'random_state' 'cross_val_type' 'k_folds' 'interval' 'mantissa'

`static Initialize`(*hyperparameters*, *dataset*)

Initializes an ANN model with the input dataset and hyperparameters.

Parameters

hyperparameters
[dict] The hyperparameters dictionary. You can create one using ANN.Hyperparameters() or, if you are using a sample Dataset, you can get it from ANN.HyperParametersBySampleName.

dataset
[sklearn.utils._bunch.Bunch] The input dataset.

Returns

_ANNModel
Returns the trained model.

`static Load`(*model*, *path*)

Loads the model state dictionary found at the input file path. The model input parameter must be pre-initialized using the ANN.Initialize() method.

Parameters

model
[ANN object] The input ANN model. The model must be pre-initialized using the ModelInitialize method.

path
[str] File path for the saved model state dictionary.

Returns

ANN model
The neural network class.

`static Metrics`(*model*)

Returns the model performance metrics given the input labels and predictions, and the model's task type.

Parameters

model
[ANN Model] The input model.

labels
[list] The input list of labels (actual values).

predictions
[list] The input list of predictions (predicted values).

Returns

dict

if the task type is 'classification', this methods return a dictionary with the following keys:
"Accuracy" "Precision" "Recall" "F1 Score" "Confusion Matrix"

else if the task type is 'regression', this method returns:
"Mean Squared Error" "Mean Absolute Error" "R-squared"

`static ModelData`(*model*)

Returns the data of the model

Parameters

model
[Model] The input model.

Returns

dict
A dictionary containing the model data. The keys in the dictionary are: 'epochs' (list of epoch numbers at which metrics data was collected) 'training_loss' (LOSS) 'validation_loss' (VALIDATION LOSS) 'training_accuracy' (ACCURACY for classification tasks only) 'validation_accuracy' (ACCURACYfor classification tasks only) 'training_mae' (MAE for regression tasks only) 'validation_mae' (MAE for regression tasks only) 'training_mse' (MSE for regression tasks only) 'validation_mse' (MSE for regression tasks only) 'training_r2' (R^2 for regression tasks only) 'validation_r2' (R^2 for regression tasks only)

`static Save`(*model*, *path*, *overwrite=False*)

Saves the model.

Parameters

model
[Model] The input model.

path
[str] The file path at which to save the model.

overwrite
[bool, optional] If set to True, any existing file will be overwritten. Otherwise, it won't. The default is False.

Returns

bool
True if the model is saved correctly. False otherwise.

`static Test`(*model*, *hyperparameters*, *dataset*)

Returns the labels (actual values) and predictions (predicted values) given the input model, features (X), and target (y).

Parameters

model
[ANN Model] The input model.

X
[list] The input list of features.

y
[list] The input list of targets

Returns

list, list
Returns two lists: metrics, and predictions.

`static Train`(*hyperparameters*, *dataset*)

Trains the input model given the input features (X), and target (y).

Parameters

hyperparameters
[dict] The hyperparameters dictionary. You can create one using ANN.Hyperparameters() or, if you are using a sample Dataset, you can get it from ANN.HyperParametersBySampleName.

dataset
[sklearn.utils._bunch.Bunch] The input dataset.

Returns

_ANNModel
Returns the trained model.

topologicpy.Aperture module

class topologicpy.Aperture.**Aperture**

Bases: object

Methods

ByTopologyContext(topology, context)	Creates an aperture object represented by the input topology and one that belongs to the input context.
Topology(aperture)	Returns the topology of the input aperture.

static ByTopologyContext(*topology*, *context*)

Creates an aperture object represented by the input topology and one that belongs to the input context.

Parameters

topology
[topologic_core.Topology] The input topology that represents the aperture.

context
[Context] The context of the aperture. See Context class.

Returns

Aperture
The created aperture.

static Topology(*aperture*)

Returns the topology of the input aperture.

Parameters

aperture
[Aperture] The input aperture.

Returns

Topology
The topology of the input aperture.

topologicpy.BVH module

class topologicpy.BVH.**BVH**

Bases: object

Methods

AABB(min_point, max_point)	

Methods

BVHNode(aabb[, left, right, objects])	
ByTopologies(*topologies[, silent])	Creates a BVH Tree from the input list of topologies.
Clashes(query)	Returns a list of topologies in the input bvh tree that clashes (broad phase) with the list of topologies in the input query.
Graph([tolerance])	Creates a graph from the input bvh tree.
MeshObject(vertices, topologic_object)	
QueryByTopologies(*topologies[, silent])	Creates a BVH Query from the input list of topologies.

class **AABB**(*min_point*, *max_point*)

Bases: object

Methods

contains	
intersects	

contains(*point*)

intersects(*other*)

class **BVHNode**(*aabb*, *left=None*, *right=None*, *objects=None*)

Bases: object

static **ByTopologies**(**topologies*, *silent: bool = False*)

Creates a BVH Tree from the input list of topologies. The input can be individual topologies each as an input argument or a list of topologies stored in one input argument.

Parameters

topologies
[list] The list of topologies.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

BVH tree
The created BVH tree.

`Clashes`(*query*)

Returns a list of topologies in the input bvh tree that clashes (broad phase) with the list of topologies in the input query.

Parameters

topologies
[list] The list of topologies.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

list
The list of clashing topologies (based on their axis-aligned bounding box (AABB))

`Graph`(*tolerance=0.0001*)

Creates a graph from the input bvh tree.

Parameters

bvh
[BVH Tree] The input BVH Tree.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The created graph.

`class MeshObject`(*vertices*, *topologic_object*)

Bases: `object`

`static QueryByTopologies`(**topologies*, *silent: bool = False*)

Creates a BVH Query from the input list of topologies. The input can be individual topologies each as an input argument or a list of topologies stored in one input argument.

Parameters

topologies
[list] The list of topologies.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

BVH query
The created BVH query.

topologicpy.Cell module

class topologicpy.Cell.**Cell**

Bases: object

Methods

Area(cell[, mantissa])	Returns the surface area of the input cell.
Box([origin, width, length, height, uSides, ...])	Creates a box.
ByFaces(faces[, planarize, tolerance, silent])	Creates a cell from the input list of faces.
ByOffset(cell[, offset, tolerance])	Creates an offset cell from the input cell.
ByShell(shell[, planarize, tolerance])	Creates a cell from the input shell.
ByShells(externalBoundary[, ...])	Creates a cell from the input external boundary (closed shell) and the input list of internal boundaries (closed shells).
ByThickenedFace(face[, thickness, ...])	Creates a cell by thickening the input face.
ByThickenedShell(shell[, direction, ...])	Creates a cell by thickening the input shell.
ByWires(wires[, close, triangulate, ...])	Creates a cell by lofting through the input list of wires.
ByWiresCluster(cluster[, close, ...])	Creates a cell by lofting through the input cluster of wires.
Capsule([origin, radius, height, uSides, ...])	Creates a capsule shape.
Compactness(cell[, reference, mantissa])	Returns the compactness measure of the input cell.
Cone([origin, baseRadius, topRadius, ...])	Creates a cone.
ContainmentStatus(cell, vertex[, tolerance])	Returns the containment status of the input vertex in relationship to the input cell
Cylinder([origin, radius, height, uSides, ...])	Creates a cylinder.
Decompose(cell[, tiltAngle, tolerance])	Decomposes the input cell into its logical components.
Dodecahedron([origin, radius, direction, ...])	Creates a dodecahedron.
Edges(cell)	Returns the edges of the input cell.
Egg([origin, height, uSides, vSides, ...])	Creates an egg-shaped cell.
ExternalBoundary(cell)	Returns the external boundary of the input cell.
Faces(cell)	Returns the faces of the input cell.
Hyperboloid([origin, baseRadius, topRadius, ...])	Creates a hyperboloid.
Icosahedron([origin, radius, direction, ...])	Creates an icosahedron.
InternalBoundaries(cell)	Returns the internal boundaries of the input cell.
InternalVertex(cell[, tolerance])	Creates a vertex that is guaranteed to be inside the input cell.
IsOnBoundary(cell, vertex[, tolerance])	Returns True if the input vertex is on the boundary of the input cell.
Octahedron([origin, radius, direction, ...])	Creates an octahedron.
Paraboloid([origin, focalLength, width, ...])	Creates a paraboloid cell.
Pipe(edge[, profile, radius, sides, ...])	Creates a pipe along the input edge.
Prism([origin, width, length, height, ...])	Creates a prism.
RemoveCollinearEdges(cell[, angTolerance, ...])	Removes any collinear edges in the input cell.
Roof(face[, angle, epsilon, tolerance])	Creates a hipped roof through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com>

continues on next page

Table 1 – continued from previous page

Sets(cells, superCells[, tolerance])	Classifies the input cells into sets based on their enclosure within the input list of super cells. The order of the sets follows the order of the input list of super cells.
Shells(cell)	Returns the shells of the input cell.
Sphere([origin, radius, uSides, vSides, ...])	Creates a sphere.
SurfaceArea(cell[, mantissa])	Returns the surface area of the input cell.
Tetrahedron([origin, radius, direction, ...])	Creates a tetrahedron.
Torus([origin, majorRadius, minorRadius, ...])	Creates a torus.
Vertices(cell)	Returns the vertices of the input cell.
Volume(cell[, mantissa])	Returns the volume of the input cell.
Wires(cell)	Returns the wires of the input cell.

static Area(*cell*, *mantissa: int = 6*)

Returns the surface area of the input cell.

Parameters

cell
[topologic_core.Cell] The cell.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The surface area of the input cell.

static Box(*origin=None*, *width: float = 1*, *length: float = 1*, *height: float = 1*, *uSides: int = 1*, *vSides: int = 1*, *wSides: int = 1*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a box.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the box. The default is None which results in the box being placed at (0, 0, 0).

width
[float , optional] The width of the box. The default is 1.

length
[float , optional] The length of the box. The default is 1.

height
[float , optional] The height of the box.

uSides
[int , optional] The number of sides along the width. The default is 1.

vSides
[int , optional] The number of sides along the length. The default is 1.

wSides
[int , optional] The number of sides along the height. The default is 1.

direction
[list , optional] The vector representing the up direction of the box. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the box. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created box.

`static ByFaces`(*faces: list*, *planarize: bool = False*, *tolerance: float = 0.0001*, *silent=False*)

Creates a cell from the input list of faces.

Parameters

faces
[list] The input list of faces.

planarize
[bool, optional] If set to True, the input faces are planarized before building the cell. Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Cell
The created cell.

`static ByOffset`(*cell*, *offset: float = 1.0*, *tolerance: float = 0.0001*)

Creates an offset cell from the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

offset
[float , optional] The desired offset distance. The default is 1.0.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

Topology
The created offset topology. WARNING: This method may fail to create a cell if the offset creates self-intersecting faces. Always check the type being returned by this method.

`static ByShell`(*shell*, *planarize: bool = False*, *tolerance: float = 0.0001*)

Creates a cell from the input shell.

Parameters

shell
[topologic_core.Shell] The input shell. The shell must be closed for this method to succeed.

planarize
[bool, optional] If set to True, the input faces of the input shell are planarized before building the cell. Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell.

`static ByShells`(*externalBoundary*, *internalBoundaries: list = []*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a cell from the input external boundary (closed shell) and the input list of internal boundaries (closed shells).

Parameters

externalBoundary
[topologic_core.Shell] The input external boundary.

internalBoundaries
[list , optional] The input list of internal boundaries (closed shells). The default is an empty list.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Cell
The created cell.

`static ByThickenedFace`(*face*, *thickness: float = 1.0*, *bothSides: bool = True*, *reverse: bool = False*, *planarize: bool = False*, *tolerance: float = 0.0001*)

Creates a cell by thickening the input face.

Parameters

face
[topologic_core.Face] The input face to be thickened.

thickness
[float , optional] The desired thickness. The default is 1.0.

bothSides
[bool] If True, the cell will be lofted to each side of the face. Otherwise, it will be lofted in the direction of the normal to the input face. The default is True.

reverse
[bool] If True, the cell will be lofted in the opposite direction of the normal to the face. The default is False.

planarize
[bool, optional] If set to True, the input faces of the input shell are planarized before building the cell. Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell.

`static ByThickenedShell`(*shell*, *direction: list = [0, 0, 1]*, *thickness: float = 1.0*, *bothSides: bool = True*, *reverse: bool = False*, *planarize: bool = False*, *tolerance: float = 0.0001*)

Creates a cell by thickening the input shell. The shell must be open.

Parameters

shell
[topologic_core.Shell] The input shell to be thickened.

thickness
[float , optional] The desired thickness. The default is 1.0.

bothSides
[bool] If True, the cell will be lofted to each side of the shell. Otherwise, it will be lofted along the input direction. The default is True.

reverse
[bool] If True, the cell will be lofted along the opposite of the input direction. The default is False.

planarize
[bool, optional] If set to True, the input faces of the input shell are planarized before building the cell. Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell.

`static ByWires`(*wires: list*, *close: bool = False*, *triangulate: bool = True*, *planarize: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*, *silent=False*)

Creates a cell by lofting through the input list of wires.

Parameters

wires
[list] The input list of wires.

close
[bool , optional] If set to True, the last wire in the list of input wires will be connected to the first wire in the list of input wires. The default is False.

triangulate
[bool , optional] If set to True, the faces will be triangulated. The default is True.

planarize
[bool, optional] If set to True, the created faces are planarized before building the cell. Otherwise, they are not. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Cell
The created cell.

Raises

Exception
Raises an exception if the two wires in the list do not have the same number of edges.

`static ByWiresCluster`(*cluster*, *close: bool = False*, *triangulate: bool = True*, *planarize: bool = False*, *tolerance: float = 0.0001*)

Creates a cell by lofting through the input cluster of wires.

Parameters

cluster
[Cluster] The input Cluster of wires.

close
[bool , optional] If set to True, the last wire in the cluster of input wires will be connected to the first wire in the cluster of input wires. The default is False.

triangulate
[bool , optional] If set to True, the faces will be triangulated. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell.

Raises

Exception
Raises an exception if the two wires in the list do not have the same number of edges.

`static Capsule`(*origin=None*, *radius: float = 0.25*, *height: float = 1*, *uSides: int = 16*, *vSidesEnds: int = 8*, *vSidesMiddle: int = 1*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a capsule shape. A capsule is a cylinder with hemispherical ends.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the cylinder. The default is None which results in the cylinder being placed at (0, 0, 0).

radius
[float , optional] The radius of the capsule. The default is 0.25.

height
[float , optional] The height of the capsule. The default is 1.

uSides
[int , optional] The number of circle segments of the capsule. The default is 16.

vSidesEnds
[int , optional] The number of vertical segments of the end hemispheres. The default is 8.

vSidesMiddle
[int , optional] The number of vertical segments of the middle cylinder. The default is 1.

direction
[list , optional] The vector representing the up direction of the capsule. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the capsule. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "bottom".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell.

`static Compactness`(*cell*, *reference='sphere'*, *mantissa: int = 6*) → float

Returns the compactness measure of the input cell. If the reference is "sphere", this is also known as 'sphericity' (https://en.wikipedia.org/wiki/Sphericity).

Parameters

cell
[topologic_core.Cell] The input cell.

reference
[str , optional] The desired reference to which to compare this compactness. The options are "sphere" and "cube". It is case insensitive. The default is "sphere".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The compactness of the input cell.

`static Cone`(*origin=None*, *baseRadius: float = 0.5*, *topRadius: float = 0*, *height: float = 1*, *uSides: int = 16*, *vSides: int = 1*, *direction: list = [0, 0, 1]*, *dirZ: float = 1*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a cone.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the cone. The default is None which results in the cone being placed at (0, 0, 0).

baseRadius
[float , optional] The radius of the base circle of the cone. The default is 0.5.

topRadius
[float , optional] The radius of the top circle of the cone. The default is 0.

height
[float , optional] The height of the cone. The default is 1.

uSides
[int , optional] The number of circle segments of the cylinder. The default is 16.

vSides
[int , optional] The number of vertical segments of the cylinder. The default is 1.

direction
[list , optional] The vector representing the up direction of the cone. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the cone. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cone.

`static ContainmentStatus`(*cell*, *vertex*, *tolerance: float = 0.0001*) → int

Returns the containment status of the input vertex in relationship to the input cell

Parameters

cell
[topologic_core.Cell] The input cell.

vertex
[topologic_core.Vertex] The input vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

int
Returns 0 if the vertex is inside, 1 if it is on the boundary of, and 2 if it is outside the input cell.

`static Cylinder`(*origin=None*, *radius: float = 0.5*, *height: float = 1*, *uSides: int = 16*, *vSides: int = 1*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a cylinder.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the cylinder. The default is None which results in the cylinder being placed at (0, 0, 0).

radius
[float , optional] The radius of the cylinder. The default is 0.5.

height
[float , optional] The height of the cylinder. The default is 1.

uSides
[int , optional] The number of circle segments of the cylinder. The default is 16.

vSides
[int , optional] The number of vertical segments of the cylinder. The default is 1.

direction
[list , optional] The vector representing the up direction of the cylinder. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the cylinder. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "bottom".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell.

`static Decompose`(*cell*, *tiltAngle: float = 10*, *tolerance: float = 0.0001*) → dict

Decomposes the input cell into its logical components. This method assumes that the positive Z direction is UP.

Parameters

cell
[topologic_core.Cell] the input cell.

tiltAngle
[float , optional] The threshold tilt angle in degrees to determine if a face is vertical, horizontal, or tilted. The tilt angle is measured from the nearest cardinal direction. The default is 10.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dictionary
A dictionary with the following keys and values: 1. "verticalFaces": list of vertical faces 2. "topHorizontalFaces": list of top horizontal faces 3. "bottomHorizontalFaces": list of bottom horizontal faces 4. "inclinedFaces": list of inclined faces 5. "verticalApertures": list of vertical apertures 6. "topHorizontalApertures": list of top horizontal apertures 7. "bottomHorizontalApertures": list of bottom horizontal apertures 8. "inclinedApertures": list of inclined apertures

`static Dodecahedron`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a dodecahedron. See https://en.wikipedia.org/wiki/Dodecahedron.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the dodecahedron. The default is None which results in the dodecahedron being placed at (0, 0, 0).

radius
[float , optional] The radius of the dodecahedron's circumscribed sphere. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the dodecahedron. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the dodecahedron. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created dodecahedron.

`static Edges`(*cell*) → list

Returns the edges of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

list
The list of edges.

`static Egg`(*origin=None*, *height: float = 1.0*, *uSides: int = 16*, *vSides: int = 8*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an egg-shaped cell.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the sphere. The default is None which results in the egg-shaped cell being placed at (0, 0, 0).

height
[float , optional] The desired height of of the egg-shaped cell. The default is 1.0.

uSides
[int , optional] The desired number of sides along the longitude of the egg-shaped cell. The default is 16.

vSides
[int , optional] The desired number of sides along the latitude of the egg-shaped cell. The default is 8.

direction
[list , optional] The vector representing the up direction of the egg-shaped cell. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the egg-shaped cell. This can be “bottom”, “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created egg-shaped cell.

`static ExternalBoundary`(*cell*)

Returns the external boundary of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

topologic_core.Shell
The external boundary of the input cell.

`static Faces`(*cell*) → list

Returns the faces of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

list
The list of faces.

`static Hyperboloid`(*origin=None*, *baseRadius: float = 0.5*, *topRadius: float = 0.5*, *height: float = 1*, *sides: int = 24*, *direction: list = [0, 0, 1]*, *twist: float = 60*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a hyperboloid.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the hyperboloid. The default is None which results in the hyperboloid being placed at (0, 0, 0).

baseRadius
[float , optional] The radius of the base circle of the hyperboloid. The default is 0.5.

topRadius
[float , optional] The radius of the top circle of the hyperboloid. The default is 0.5.

height
[float , optional] The height of the cone. The default is 1.

sides
[int , optional] The number of sides of the cone. The default is 24.

direction
[list , optional] The vector representing the up direction of the hyperboloid. The default is [0, 0, 1].

twist
[float , optional] The angle to twist the base cylinder. The default is 60.

placement
[str , optional] The description of the placement of the origin of the hyperboloid. This can be “bottom”, “center”, or “lowerleft”. It is case insensitive. The default is “center”.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created hyperboloid.

`static Icosahedron`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an icosahedron. See https://en.wikipedia.org/wiki/Icosahedron.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the icosahedron. The default is None which results in the icosahedron being placed at (0, 0, 0).

radius
[float , optional] The radius of the icosahedron’s circumscribed sphere. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the icosahedron. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the icosahedron. This can be “bottom”, “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created icosahedron.

`static InternalBoundaries`(*cell*) → list

Returns the internal boundaries of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

list
The list of internal boundaries ([topologic_core.Shell]).

`static InternalVertex`(*cell*, *tolerance: float = 0.0001*)

Creates a vertex that is guaranteed to be inside the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Vertex
The internal vertex.

`static IsOnBoundary`(*cell*, *vertex*, *tolerance: float = 0.0001*) → bool

Returns True if the input vertex is on the boundary of the input cell. Returns False otherwise.

Parameters

cell
[topologic_core.Cell] The input cell.

vertex
[topologic_core.Vertex] The input vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
Returns True if the input vertex is inside the input cell. Returns False otherwise.

`static Octahedron`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an octahedron. See https://en.wikipedia.org/wiki/Octahedron.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the octahedron. The default is None which results in the octahedron being placed at (0, 0, 0).

radius
[float , optional] The radius of the octahedron's circumscribed sphere. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the octahedron. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the octahedron. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created octahedron.

`static Paraboloid`(*origin=None*, *focalLength=0.125*, *width: float = 1*, *length: float = 1*, *height: float = 0*, *uSides: int = 16*, *vSides: int = 16*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*, *silent=False*)

Creates a paraboloid cell. See https://en.wikipedia.org/wiki/Paraboloid

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the parabolic surface. The default is None which results in the parabolic surface being placed at (0, 0, 0).

focalLength
[float , optional] The focal length of the parabola. The default is 0.125.

width
[float , optional] The width of the parabolic surface. The default is 1.

length
[float , optional] The length of the parabolic surface. The default is 1.

height
[float , optional] The additional height of the parabolic surface. Please note this is not the height from the spring point to the apex. It is in addition to that to form a base. The default is 0.

uSides
[int , optional] The number of sides along the width. The default is 16.

vSides
[int , optional] The number of sides along the length. The default is 16.

direction
[list , optional] The vector representing the up direction of the parabolic surface. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the parabolic surface. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional]

If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Shell
The created paraboloid.

`static Pipe`(*edge*, *profile=None*, *radius: float = 0.5*, *sides: int = 16*, *startOffset: float = 0*, *endOffset: float = 0*, *endcapA=None*, *endcapB=None*, *mantissa: int = 6*) → dict

Creates a pipe along the input edge.

Parameters

edge
[topologic_core.Edge] The centerline of the pipe.

profile
[topologic_core.Wire , optional] The profile of the pipe. It is assumed that the profile is in the XY plane. If set to None, a circle of radius 0.5 will be used. The default is None.

radius
[float , optional] The radius of the pipe. The default is 0.5.

sides
[int , optional] The number of sides of the pipe. The default is 16.

startOffset
[float , optional] The offset distance from the start vertex of the centerline edge. The default is 0.

endOffset
[float , optional] The offset distance from the end vertex of the centerline edge. The default is 0.

endcapA, optional
The topology to place at the start vertex of the centerline edge. The positive Z direction of the end cap will be oriented in the direction of the centerline edge.

endcapB, optional
The topology to place at the end vertex of the centerline edge. The positive Z direction of the end cap will be oriented in the inverse direction of the centerline edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

Returns

dict
A dictionary containing the pipe, the start endcap, and the end endcap if they have been specified. The dictionary has the following keys: 'pipe' 'endcapA' 'endcapB'

`static Prism`(*origin=None*, *width: float = 1*, *length: float = 1*, *height: float = 1*, *uSides: int = 1*, *vSides: int = 1*, *wSides: int = 1*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a prism.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the prism. The default is None which results in the prism being placed at (0, 0, 0).

width
[float , optional] The width of the prism. The default is 1.

length
[float , optional] The length of the prism. The default is 1.

height
[float , optional] The height of the prism.

uSides
[int , optional] The number of sides along the width. The default is 1.

vSides
[int , optional] The number of sides along the length. The default is 1.

wSides
[int , optional] The number of sides along the height. The default is 1.

direction
[list , optional] The vector representing the up direction of the prism. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the prism. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created prism.

`static RemoveCollinearEdges`(*cell*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Removes any collinear edges in the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created cell without any collinear edges.

`static Roof`(*face*, *angle: float = 45*, *epsilon: float = 0.01*, *tolerance: float = 0.001*)

Creates a hipped roof through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com> This algorithm depends on the polyskel code which is included in the library. Polyskel code is found at: https://github.com/Botffy/polyskel

Parameters

face
[topologic_core.Face] The input face.

angle
[float , optional] The desired angle in degrees of the roof. The default is 45.

epsilon
[float , optional] The desired epsilon (another form of tolerance for distance from plane). The default is 0.01. (This is set to a larger number as it was found to work better)

tolerance
[float , optional] The desired tolerance. The default is 0.001. (This is set to a larger number as it was found to work better)

Returns

cell
The created roof.

static Sets(*cells: list*, *superCells: list*, *tolerance: float = 0.0001*) → list

Classifies the input cells into sets based on their enclosure within the input list of super cells. The order of the sets follows the order of the input list of super cells.

Parameters

inputCells
[list] The list of input cells.

superCells
[list] The list of super cells.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The classified list of input cells based on their enclosure within the input list of super cells.

static Shells(*cell*) → list

Returns the shells of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

list
The list of shells.

static Sphere(*origin=None*, *radius: float = 0.5*, *uSides: int = 16*, *vSides: int = 8*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a sphere.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the sphere. The default is None which results in the sphere being placed at (0, 0, 0).

radius
[float , optional] The radius of the sphere. The default is 0.5.

uSides
[int , optional] The number of sides along the longitude of the sphere. The default is 16.

vSides
[int , optional] The number of sides along the latitude of the sphere. The default is 8.

direction
[list , optional] The vector representing the up direction of the sphere. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the sphere. This can be “bottom”, “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created sphere.

`static SurfaceArea`(*cell*, *mantissa: int = 6*) → float

Returns the surface area of the input cell.

Parameters

cell
[topologic_core.Cell] The cell.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

area
[float] The surface area of the input cell.

`static Tetrahedron`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a tetrahedron. See https://en.wikipedia.org/wiki/Tetrahedron.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the tetrahedron. The default is None which results in the tetrahedron being placed at (0, 0, 0).

radius
[float , optional] The radius of the tetrahedron's circumscribed sphere. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the tetrahedron. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the tetrahedron. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created tetrahedron.

`static Torus`(*origin=None*, *majorRadius: float = 0.5*, *minorRadius: float = 0.125*, *uSides: int = 16*, *vSides: int = 8*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a torus.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the torus. The default is None which results in the torus being placed at (0, 0, 0).

majorRadius
[float , optional] The major radius of the torus. The default is 0.5.

minorRadius
[float , optional] The minor radius of the torus. The default is 0.1.

uSides
[int , optional] The number of sides along the longitude of the torus. The default is 16.

vSides
[int , optional] The number of sides along the latitude of the torus. The default is 8.

direction
[list , optional] The vector representing the up direction of the torus. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the torus. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created torus.

`static Vertices`(*cell*) → list

Returns the vertices of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

list
The list of vertices.

`static Volume`(*cell*, *mantissa: int = 6*) → float

Returns the volume of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

manitssa: int , optional
The desired length of the mantissa. The default is 6.

Returns

float
The volume of the input cell.

`static Wires`(*cell*) → list

Returns the wires of the input cell.

Parameters

cell
[topologic_core.Cell] The input cell.

Returns

list
The list of wires.

topologicpy.CellComplex module

class topologicpy.CellComplex.**CellComplex**

Bases: object

Methods

Box([origin, width, length, height, uSides, ...])	Creates a box with internal cells.
ByCells(cells[, transferDictionaries, ...])	Creates a cellcomplex by merging the input cells.
ByCellsCluster(cluster[, tolerance])	Creates a cellcomplex by merging the cells within the input cluster.
ByFaces(faces[, tolerance, silent])	Creates a cellcomplex by merging the input faces.
ByFacesCluster(cluster[, tolerance])	Creates a cellcomplex by merging the faces within the input cluster.
ByWires(wires[, triangulate, tolerance])	Creates a cellcomplex by lofting through the input wires.
ByWiresCluster(cluster[, triangulate, tolerance])	Creates a cellcomplex by lofting through the wires in the input cluster.
Cells(cellComplex)	Returns the cells of the input cellComplex.
Decompose(cellComplex[, tiltAngle, tolerance])	Decomposes the input cellComplex into its logical components.
Delaunay([vertices, tolerance])	Triangulates the input vertices based on the Delaunay method.
Edges(cellComplex)	Returns the edges of the input cellComplex.
ExternalBoundary(cellComplex)	Returns the external boundary (cell) of the input cellComplex.
ExternalFaces(cellComplex)	Returns the external faces of the input cellComplex.
Faces(cellComplex)	Returns the faces of the input cellComplex.
InternalFaces(cellComplex)	Returns the internal boundaries (faces) of the input cellComplex.
NonManifoldFaces(cellComplex)	Returns the non-manifold faces of the input cellComplex.
Octahedron([origin, radius, direction, ...])	Creates an octahedron.
Prism([origin, width, length, height, ...])	Creates a prismatic cellComplex with internal cells.
RemoveCollinearEdges(cellComplex[, ...])	Removes any collinear edges in the input cellComplex.
Shells(cellComplex)	Returns the shells of the input cellComplex.
Torus([origin, majorRadius, minorRadius, ...])	Creates a torus.
Vertices(cellComplex)	Returns the vertices of the input cellComplex.
Volume(cellComplex[, mantissa])	Returns the volume of the input cellComplex.
Voronoi([vertices, cell, tolerance])	Partitions the input cell based on the Voronoi method.
Wires(cellComplex)	Returns the wires of the input cellComplex.

static Box(*origin=None*, *width: float = 1.0*, *length: float = 1.0*, *height: float = 1.0*, *uSides: int = 2*, *vSides: int = 2*, *wSides: int = 2*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a box with internal cells.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the box. The default is None which results in the box being placed at (0, 0, 0).

width
[float , optional] The width of the box. The default is 1.

length
[float , optional] The length of the box. The default is 1.

height
[float , optional] The height of the box.

uSides
[int , optional] The number of sides along the width. The default is 1.

vSides
[int, optional] The number of sides along the length. The default is 1.

wSides
[int , optional] The number of sides along the height. The default is 1.

direction
[list , optional] The vector representing the up direction of the box. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the box. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created box.

`static ByCells`(*cells: list*, *transferDictionaries=False*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a cellcomplex by merging the input cells.

Parameters

cells
[list] The list of input cells.

transferDictionaries
[bool , optional] If set to True, any dictionaries in the cells are transferred to the CellComplex. Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.CellComplex
The created cellcomplex.

`static ByCellsCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a cellcomplex by merging the cells within the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster of cells.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created cellcomplex.

`static ByFaces`(*faces: list*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a cellcomplex by merging the input faces.

Parameters

faces
[list] The input faces.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.CellComplex
The created cellcomplex.

`static ByFacesCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a cellcomplex by merging the faces within the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster of faces.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created cellcomplex.

`static ByWires`(*wires: list*, *triangulate: bool = True*, *tolerance: float = 0.0001*)

Creates a cellcomplex by lofting through the input wires.

Parameters

wires
[list] The input list of wires. The list should contain a minimum of two wires. All wires must have the same number of edges.

triangulate
[bool , optional] If set to True, the faces will be triangulated. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created cellcomplex.

`static ByWiresCluster`(*cluster*, *triangulate: bool = True*, *tolerance: float = 0.0001*)

Creates a cellcomplex by lofting through the wires in the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster of wires.

triangulate
[bool , optional] If set to True, the faces will be triangulated. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created cellcomplex.

`static Cells`(*cellComplex*) → list

Returns the cells of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of cells.

`static Decompose`(*cellComplex*, *tiltAngle: float = 10.0*, *tolerance: float = 0.0001*) → dict

Decomposes the input cellComplex into its logical components. This method assumes that the positive Z direction is UP.

Parameters

cellComplex
[topologic_core.CellComplex] the input cellComplex.

tiltAngle
[float , optional] The threshold tilt angle in degrees to determine if a face is vertical, horizontal, or tilted. The tilt angle is measured from the nearest cardinal direction. The default is 10.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dictionary
A dictionary with the following keys and values: 1. “cells”: list of cells 2. “externalVerticalFaces”: list of external vertical faces 3. “internalVerticalFaces”: list of internal vertical faces 4. “topHorizontalFaces”: list of top horizontal faces 5. “bottomHorizontalFaces”: list of bottom horizontal faces 6. “internalHorizontalFaces”: list of internal horizontal

faces 7. “externalInclinedFaces”: list of external inclined faces 8. “internalInclinedFaces”: list of internal inclined faces 9. “externalVerticalApertures”: list of external vertical apertures 10. “internalVerticalApertures”: list of internal vertical apertures 11. “topHorizontalApertures”: list of top horizontal apertures 12. “bottomHorizontalApertures”: list of bottom horizontal apertures 13. “internalHorizontalApertures”: list of internal horizontal apertures 14. “externalInclinedApertures”: list of external inclined apertures 15. “internalInclinedApertures”: list of internal inclined apertures

static Delaunay(*vertices: list = None*, *tolerance: float = 0.0001*)

Triangulates the input vertices based on the Delaunay method. See https://en.wikipedia.org/wiki/Delaunay_triangulation.

Parameters

vertices: list , optional
The input list of vertices to use for delaunay triangulation. If set to None, the algorithm uses the vertices of the input cell parameter. if both are set to none, a unit cube centered around the origin is used.

tolerance
[float , optional] the desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created delaunay cellComplex.

static Edges(*cellComplex*) → list

Returns the edges of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of edges.

static ExternalBoundary(*cellComplex*)

Returns the external boundary (cell) of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

topologic_core.Cell
The external boundary of the input cellComplex.

static ExternalFaces(*cellComplex*) → list

Returns the external faces of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of external faces.

`static Faces`(*cellComplex*) → list

Returns the faces of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of faces.

`static InternalFaces`(*cellComplex*) → list

Returns the internal boundaries (faces) of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of internal faces of the input cellComplex.

`static NonManifoldFaces`(*cellComplex*) → list

Returns the non-manifold faces of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of non-manifold faces of the input cellComplex.

`static Octahedron`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an octahedron. See https://en.wikipedia.org/wiki/Octahedron.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the octahedron. The default is None which results in the octahedron being placed at (0, 0, 0).

radius
[float , optional] The radius of the octahedron's circumscribed sphere. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the octahedron. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the octahedron. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created octahedron.

`static Prism`(*origin=None*, *width: float = 1.0*, *length: float = 1.0*, *height: float = 1.0*, *uSides: int = 2*, *vSides: int = 2*, *wSides: int = 2*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a prismatic cellComplex with internal cells.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the prism. The default is None which results in the prism being placed at (0, 0, 0).

width
[float , optional] The width of the prism. The default is 1.

length
[float , optional] The length of the prism. The default is 1.

height
[float , optional] The height of the prism.

uSides
[int , optional] The number of sides along the width. The default is 1.

vSides
[int , optional] The number of sides along the length. The default is 1.

wSides
[int , optional] The number of sides along the height. The default is 1.

direction
[list , optional] The vector representing the up direction of the prism. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the prism. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created prism.

`static RemoveCollinearEdges`(*cellComplex*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Removes any collinear edges in the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created cellComplex without any collinear edges.

`static Shells`(*cellComplex*) → list

Returns the shells of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of shells.

`static Torus`(*origin=None*, *majorRadius: float = 0.5*, *minorRadius: float = 0.125*, *uSides: int = 16*, *vSides: int = 8*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a torus.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the torus. The default is None which results in the torus being placed at (0, 0, 0).

majorRadius
[float , optional] The major radius of the torus. The default is 0.5.

minorRadius
[float , optional] The minor radius of the torus. The default is 0.1.

uSides
[int , optional] The number of sides along the longitude of the torus. The default is 16.

vSides
[int , optional] The number of sides along the latitude of the torus. The default is 8.

direction
[list , optional] The vector representing the up direction of the torus. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the torus. This can be “bottom”, “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell
The created torus.

`static Vertices`(*cellComplex*) → list

Returns the vertices of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of vertices.

`static Volume`(*cellComplex*, *mantissa: int = 6*) → float

Returns the volume of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

manitssa: int , optional
The desired length of the mantissa. The default is 6.

Returns

float
The volume of the input cellComplex.

`static Voronoi`(*vertices: list = None*, *cell=None*, *tolerance: float = 0.0001*)

Partitions the input cell based on the Voronoi method. See https://en.wikipedia.org/wiki/Voronoi_diagram.

Parameters

vertices: list , optional
The input list of vertices to use for voronoi partitioning. If set to None, the algorithm uses the vertices of the input cell parameter. if both are set to none, a unit cube centered around the origin is used.

cell
[topologic_core.Cell , optional] The input bounding cell. If set to None, an axes-aligned bounding cell is created from the list of vertices. The default is None.

tolerance
[float , optional] the desired tolerance. The default is 0.0001.

Returns

topologic_core.CellComplex
The created voronoi cellComplex.

`static Wires`(*cellComplex*) → list

Returns the wires of the input cellComplex.

Parameters

cellComplex
[topologic_core.CellComplex] The input cellComplex.

Returns

list
The list of wires.

topologicpy.Cluster module

class topologicpy.Cluster.**Cluster**

Bases: object

Methods

ByFormula(formula[, xRange, yRange, ...])	Creates a cluster of vertices by evaluating the input formula for a range of x values and, optionally, a range of y values.
ByTopologies(*args[, transferDictionaries, ...])	Creates a topologic Cluster from the input list of topologies.
CellComplexes(cluster)	Returns the cellComplexes of the input cluster.
Cells(cluster)	Returns the cells of the input cluster.
DBSCAN(topologies[, selectors, keys, ...])	Clusters the input vertices based on the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) method.
Edges(cluster)	Returns the edges of the input cluster.
Faces(cluster)	Returns the faces of the input cluster.
FreeCells(cluster[, tolerance])	Returns the free cells of the input cluster that are not part of a higher topology.
FreeEdges(cluster[, tolerance])	Returns the free edges of the input cluster that are not part of a higher topology.
FreeFaces(cluster[, tolerance])	Returns the free faces of the input cluster that are not part of a higher topology.
FreeShells(cluster[, tolerance])	Returns the free shells of the input cluster that are not part of a higher topology.
FreeTopologies(cluster[, tolerance])	Returns the free topologies of the input cluster that are not part of a higher topology.
FreeVertices(cluster[, tolerance])	Returns the free vertices of the input cluster that are not part of a higher topology.
FreeWires(cluster[, tolerance])	Returns the free wires of the input cluster that are not part of a higher topology.
HighestType(cluster)	Returns the type of the highest dimension subtopology found in the input cluster.
K_Means(topologies[, selectors, keys, k, ...])	Clusters the input topologies using K-Means clustering.
MergeCells(cells[, tolerance])	Creates a cluster that contains cellComplexes where it can create them plus any additional free cells.
MysticRose([wire, origin, radius, sides, ...])	Creates a mystic rose.
Shells(cluster)	Returns the shells of the input cluster.
Simplify(cluster)	Simplifies the input cluster if possible.
Vertices(cluster)	Returns the vertices of the input cluster.
Wires(cluster)	Returns the wires of the input cluster.

static ByFormula(*formula*, *xRange=None*, *yRange=None*, *xString='X'*, *yString='Y'*)

Creates a cluster of vertices by evaluating the input formula for a range of x values and, optionally, a range of y values.

Parameters

formula
[str] A string representing the formula to be evaluated. For 2D formulas (i.e. Z = 0), use either 'X' (uppercase) or 'Y' (uppercase) for the independent variable. For 3D formulas, use 'X' and 'Y' (uppercase) for the independent variables. The Z value will be evaluated. For 3D formulas, both xRange and yRange MUST be specified. You can use standard math functions like 'sin', 'cos', 'tan', 'sqrt', etc. For example, 'X**2 + 2*X - sqrt(X)' or 'cos(abs(X)+abs(Y))'

xRange
[tuple , optional] A tuple (start, end, step) representing the range of X values for which the formula should be evaluated. For example, to evaluate Y for X values from -5 to 5 with a step of 0.1, you should specify xRange=(-5, 5, 0.1). If the xRange is set to None or not specified:

The method assumes that the formula uses the yString (e.g. 'Y' as in 'Y**2 + 2*Y - sqrt(Y)') The method will attempt to evaluate X based on the specified yRange. xRange and yRange CANNOT be None or unspecified at the same time. One or the other must be specified.

yRange
[tuple , optional] A tuple (start, end, step) representing the range of Y values for which the formula should be evaluated. For example, to evaluate X for Y values from -5 to 5 with a step of 0.1, you should specify yRange=(-5,5,0.1). If the yRange is set to None or not specified:

The method assumes that the formula uses the xString (e.g. 'X' as in 'X**2 + 2*X - sqrt(X)') The method will attempt to evaluate Y based on the specified xRange. xRange and yRange CANNOT be None or unspecified at the same time. One or the other must be specified.

xString
[str , optional] The string used to represent the X independent variable. The default is 'X' (uppercase).

yString
[str , optional] The string used to represent the Y independent variable. The default is 'Y' (uppercase).

Returns:

topologic_core.Cluster
The created cluster of vertices.

`static ByTopologies`(**args*, *transferDictionaries: bool = False*, *silent=False*)

Creates a topologic Cluster from the input list of topologies. The input can be individual topologies each as an input argument or a list of topologies stored in one input argument.

Parameters

topologies
[list] The list of topologies.

transferDictionaries
[bool , optional] If set to True, the dictionaries from the input topologies are merged and transferred to the cluster. Otherwise they are not. The default is False.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Cluster
The created topologic Cluster.

`static CellComplexes`(*cluster*) → list

Returns the cellComplexes of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of cellComplexes.

`static Cells`(*cluster*) → list

Returns the cells of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of cells.

`static DBSCAN`(*topologies*, *selectors=None*, *keys=['x', 'y', 'z']*, *epsilon: float = 0.5*, *minSamples: int = 2*)

Clusters the input vertices based on the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) method. See https://en.wikipedia.org/wiki/DBSCAN

Parameters

topologies
[list] The input list of topologies to be clustered.

selectors
[list , optional] If the list of topologies are not vertices then please provide a corresponding list of selectors (vertices) that represent the topologies for clustering. For example, these can be the centroids of the topologies. If set to None, the list of topologies is expected to be a list of vertices. The default is None.

keys
[list, optional] The keys in the embedded dictionaries in the topologies. If specified, the values at these keys will be added to the dimensions to be clustered. The values must be numeric. If you wish the x, y, z location to be included, make sure the keys list includes "X", "Y", and/or "Z" (case insensitive). The default is ["x", "y", "z"]

epsilon
[float , optional] The maximum radius around a data point within which other points are considered to be part of the same sense region (cluster). The default is 0.5.

minSamples
[int , optional] The minimum number of points required to form a dense region (cluster). The default is 2.

Returns

list, list
The list of clusters and the list of vertices considered to be noise if any (otherwise returns None).

`static Edges`(*cluster*) → list

Returns the edges of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of edges.

`static Faces`(*cluster*) → list

Returns the faces of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of faces.

`static FreeCells`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free cells of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free cells.

`static FreeEdges`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free edges of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free edges.

`static FreeFaces`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free faces of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free faces.

`static FreeShells`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free shells of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free shells.

`static FreeTopologies`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free topologies of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free topologies.

`static FreeVertices`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free vertices of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free vertices.

`static FreeWires`(*cluster*, *tolerance: float = 0.0001*) → list

Returns the free wires of the input cluster that are not part of a higher topology.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of free wires.

`static HighestType`(*cluster*) → int

Returns the type of the highest dimension subtopology found in the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

int
The type of the highest dimension subtopology found in the input cluster.

`static K_Means`(*topologies*, *selectors=None*, *keys=['x', 'y', 'z']*, *k=4*, *maxIterations=100*, *centroidKey='k_centroid'*)

Clusters the input topologies using K-Means clustering. See https://en.wikipedia.org/wiki/K-means_clustering

Parameters

topologies
[list] The input list of topologies. If this is not a list of topologic vertices then please provide a list of selectors

selectors
[list , optional] If the list of topologies are not vertices then please provide a corresponding list of selectors (vertices) that represent the topologies for clustering. For example, these can be the centroids of the topologies. If set to None, the list of topologies is expected to be a list of vertices. The default is None.

keys
[list, optional] The keys in the embedded dictionaries in the topologies. If specified, the values at these keys will be added to the dimensions to be clustered. The values must be numeric. If you wish the x, y, z location to be included, make sure the keys list includes "X", "Y", and/or "Z" (case insensitive). The default is ["x", "y", "z"]

k
[int , optional] The desired number of clusters. The default is 4.

maxIterations
[int , optional] The desired maximum number of iterations for the clustering algorithm

centroidKey
[str , optional] The desired dictionary key under which to store the cluster's centroid (this is not to be confused with the actual geometric centroid of the cluster). The default is "k_centroid"

Returns

list
The created list of clusters.

`static MergeCells`(*cells*, *tolerance=0.0001*)

Creates a cluster that contains cellComplexes where it can create them plus any additional free cells.

Parameters

cells
[list] The input list of cells.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The created cluster with merged cells as possible.

`static MysticRose`(*wire=None*, *origin=None*, *radius: float = 0.5*, *sides: int = 16*, *perimeter: bool = True*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a mystic rose.

Parameters

wire
[topologic_core.Wire , optional] The input Wire. if set to None, a circle with the input parameters is created. Otherwise, the input parameters are ignored.

origin
[topologic_core.Vertex , optional] The location of the origin of the circle. The default is None which results in the circle being placed at (0, 0, 0).

radius
[float , optional] The radius of the mystic rose. The default is 1.

sides
[int , optional] The number of sides of the mystic rose. The default is 16.

perimeter
[bool , optional] If True, the perimeter edges are included in the output. The default is True.

direction
[list , optional] The vector representing the up direction of the mystic rose. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the mystic rose. This can be “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The created mystic rose (cluster of edges).

`static Shells`(*cluster*) → list

Returns the shells of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of shells.

`static Simplify`(*cluster*)

Simplifies the input cluster if possible. For example, if the cluster contains only one cell, that cell is returned.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

topologic_core.Topology or list
The simplification of the cluster.

`static Vertices`(*cluster*) → list

Returns the vertices of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of vertices.

`static Wires`(*cluster*) → list

Returns the wires of the input cluster.

Parameters

cluster
[topologic_core.Cluster] The input cluster.

Returns

list
The list of wires.

topologicpy.Color module

`class topologicpy.Color.Color`

Bases: `object`

Methods

AnyToHex(color)	Converts a color to a hexadecimal color string.
ByCSSNamedColor(color[, alpha])	Creates a Color from a CSS named color string.
ByHEX(hex[, alpha])	Converts a hexadecimal color string to RGB color values.
ByValueInRange([value, minValue, maxValue, ...])	Returns the r, g, b, (and optionally) a list of numbers representing the red, green, blue and alpha color elements.
CMYKToHex(cmyk)	Convert a CMYK color (list of 4 values) to its hexadecimal representation.
CSSNamedColor(color)	Returns the CSS Named color that most closely matches the input color.
CSSNamedColors()	Returns a list of all CSS named colors.
PlotlyColor(color[, alpha, useAlpha])	Returns a plotly color string based on the input list of [r, g, b] or [r, g, b, a].
RGBToHex(rgb)	Converts RGB color values to a hexadecimal color string.

static AnyToHex(*color*)

Converts a color to a hexadecimal color string.

Parameters

color
[list or str] The input color parameter which can be any of RGB, CMYK, CSS Named Color, or Hex

Returns

str
A hexadecimal color string in the format '#RRGGBB'.

static ByCSSNamedColor(*color*, *alpha: float = None*)

Creates a Color from a CSS named color string. See https://developer.mozilla.org/en-US/docs/Web/CSS/named-color

Parameters

color
[str] A CSS named color.

alpha
[float , optional] THe desired alpha (transparency value). The default is None which means no alpha value will be included in the returned list.

Returns

list
The color expressed as an [r, g, b] or an [r, g, b, a] list.

static ByHEX(*hex: str*, *alpha: float = None*)

Converts a hexadecimal color string to RGB color values.

Parameters

hex
[str] A hexadecimal color string in the format '#RRGGBB'.

alpha
[float , optional] The transparency value. 0.0 means the color is fully transparent, 1.0 means the color is fully opaque. The default is None which means no transparency value will be included in the returned color.

Returns
——-
list
The color expressed as an [r, g, b] or an [r, g, b, a] list.

`static ByValueInRange`(*value: float = 0.5*, *minValue: float = 0.0*, *maxValue: float = 1.0*, *alpha: float = None*, *colorScale='viridis'*)

Returns the r, g, b, (and optionally) a list of numbers representing the red, green, blue and alpha color elements.

Parameters

value
[float , optional] The input value. The default is 0.5.

minValue
[float , optional] the input minimum value. The default is 0.0.

maxValue
[float , optional] The input maximum value. The default is 1.0.

alpha
[float , optional] The alpha (transparency) value. 0.0 means the color is fully transparent, 1.0 means the color is fully opaque. The default is 1.0.

useAlpha
[bool , optional] If set to True, the returns list includes the alpha value as a fourth element in the list.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. “Viridis”, “Plasma”). The default is “Viridis”. For a full list of names, see https://plotly.com/python/builtin-colorscales/.

Returns

list
The color expressed as an [r, g, b] or an [r, g, b, a] list.

`static CMYKToHex`(*cmyk*)

Convert a CMYK color (list of 4 values) to its hexadecimal representation.

Parameters

color
[list] cmyk (list or tuple): CMYK color values as [C, M, Y, K], each in the range 0 to 1.

Returns

str: The hexadecimal color string for Plotly (e.g., ‘#FFFFFF’).

`static CSSNamedColor`(*color*)

Returns the CSS Named color that most closely matches the input color. The input color is assumed to be in the format [r, g, b]. See https://developer.mozilla.org/en-US/docs/Web/CSS/named-color

Parameters

color
[list] The input color. This is assumed to be in the format [r, g, b]

Returns

str
The CSS named color that most closely matches the input color.

`static CSSNamedColors()`

Returns a list of all CSS named colors. See https://developer.mozilla.org/en-US/docs/Web/CSS/named-color

Parameters

Returns

list
The list of all CSS named colors.

`static PlotlyColor(color, alpha=1.0, useAlpha=False)`

Returns a plotly color string based on the input list of [r, g, b] or [r, g, b, a]. If your list is [r, g, b], you can optionally specify an alpha value

Parameters

color
[list] The input color list. This is assumed to be in the format [r, g, b] or [r, g, b, a] where the range is from 0 to 255.

alpha
[float , optional] The transparency value. 0.0 means the color is fully transparent, 1.0 means the color is fully opaque. The default is 1.0.

useAlpha
[bool , optional] If set to True, the returns list includes the alpha value as a fourth element in the list.

Returns

str
The plotly color string.

`static RGBToHex(rgb)`

Converts RGB color values to a hexadecimal color string.

Parameters

rgb
[tuple] A tuple containing three integers representing the RGB values.

Returns

str
A hexadecimal color string in the format '#RRGGBB'.

topologicpy.Context module

class topologicpy.Context.**Context**

Bases: object

Methods

ByTopologyParameters(topology[, u, v, w])	Creates a context object represented by the input topology.
Topology(context)	Returns the topology of the input context.

static ByTopologyParameters(*topology*, *u=0.5*, *v=0.5*, *w=0.5*)

Creates a context object represented by the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

u
[float , optional] The input *u* parameter. This defines the relative parameteric location of the content object along the *u* axis.

v
[TYPE] The input *v* parameter. This defines the relative parameteric location of the content object along the *v* axis..

w
[TYPE] The input *w* parameter. This defines the relative parameteric location of the content object along the *w* axis.

Returns

topologic_core.Context
The created context object. See Aperture.ByObjectContext.

static Topology(*context*)

Returns the topology of the input context.

Parameters

context
[topologic_core.Context] The input context.

Returns

topologic_core.Topology
The topology of the input context.

topologicpy.DGL module

topologicpy.Dictionary module

class topologicpy.Dictionary.**Dictionary**

Bases: object

@staticmethod def ByDGLData(item):

“”” Parameters ———- item : TYPE

DESCRIPTION.

Returns

dictionaries
[TYPE] DESCRIPTION.

“”” **keys = list(item.keys())**
vList = []
for k in keys:
vList.append(item[k].tolist())

dictionaries = []
for v in range(len(vList[0])):
values = [] for k in range(len(keys)):

value = vList[k][v] values.append(value)

dictionaries.append(Dictionary.ByKeysValues(keys, values))

return dictionaries

Methods

ByKeyValue(key, value)	Creates a Dictionary from the input key and the input value.
ByKeysValues(keys, values)	Creates a Dictionary from the input list of keys and the input list of values.
ByMergedDictionaries(*dictionaries[, silent])	Creates a dictionary by merging the list of input dictionaries.
ByPythonDictionary(pythonDictionary)	Creates a dictionary equivalent to the input python dictionary.
Filter(elements, dictionaries[, searchType, ...])	Filters the input list of dictionaries based on the input parameters.
Keys(dictionary)	Returns the keys of the input dictionary.
ListAttributeValues(listAttribute)	Returns the list of values embedded in the input listAttribute.
PythonDictionary(dictionary)	Returns the input dictionary as a python dictionary
RemoveKey(dictionary, key)	Removes the key (and its associated value) from the input dictionary.
SetValueAtKey(dictionary, key, value)	Creates a key/value pair in the input dictionary.
ValueAtKey(dictionary, key[, silent])	Returns the value of the input key in the input dictionary.
Values(dictionary)	Returns the list of values in the input dictionary.

`static ByKeyValue`(*key*, *value*)

Creates a Dictionary from the input key and the input value.

Parameters

key
[str] The string representing the key of the value in the dictionary.

value
[int, float, str, or list] A value corresponding to the input key. A value can be an integer, a float, a string, or a list.

Returns

topologic_core.Dictionary
The created dictionary.

`static ByKeysValues`(*keys*, *values*)

Creates a Dictionary from the input list of keys and the input list of values.

Parameters

keys
[list] A list of strings representing the keys of the dictionary.

values
[list] A list of values corresponding to the list of keys. Values can be integers, floats, strings, or lists

Returns

topologic_core.Dictionary
The created dictionary.

`static ByMergedDictionaries`(**dictionaries*, *silent: bool = False*)

Creates a dictionary by merging the list of input dictionaries.

Parameters

dictionaries
[list or comma separated dictionaries] The input list of dictionaries to be merged.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Dictionary
The created dictionary.

`static ByPythonDictionary`(*pythonDictionary*)

Creates a dictionary equivalent to the input python dictionary.

Parameters

pythonDictionary
[dict] The input python dictionary.

Returns

topologic_core.Dictionary
The dictionary equivalent to the input python dictionary.

`static Filter`(*elements*, *dictionaries*, *searchType='any'*, *key=None*, *value=None*)

Filters the input list of dictionaries based on the input parameters.

Parameters

elements
[list] The input list of elements to be filtered according to the input dictionaries.

dictionaries
[list] The input list of dictionaries to be filtered.

searchType
[str , optional] The type of search query to conduct in the topology's dictionary. This can be one of "any", "equal to", "contains", "starts with", "ends with", "not equal to", "does not contain". The default is "any".

key
[str , optional] The dictionary key to search within. The default is None which means it will filter by topology type only.

value
[str , optional] The value to search for at the specified key. The default is None which means it will filter by topology type only.

Returns

dict
A dictionary of filtered and other elements. The dictionary has two keys: - "filtered" The filtered dictionaries. - "other" The other dictionaries that did not meet the filter criteria. - "filteredIndices" The filtered indices of dictionaries. - "otherIndices" The other indices of dictionaries that did not meet the filter criteria.

`static Keys`(*dictionary*)

Returns the keys of the input dictionary.

Parameters

dictionary
[topologic_core.Dictionary or dict] The input dictionary.

Returns

list
The list of keys of the input dictionary.

`static ListAttributeValues`(*listAttribute*)

Returns the list of values embedded in the input listAttribute.

Parameters

listAttribute
[listAttribute] The input list attribute.

Returns

list
The list of values found in the input list attribute

`static PythonDictionary`(*dictionary*)

Returns the input dictionary as a python dictionary

Parameters

dictionary
[topologic_core.Dictionary] The input dictionary.

Returns

dict
The python dictionary equivalent of the input dictionary

`static RemoveKey`(*dictionary*, *key*)

Removes the key (and its associated value) from the input dictionary.

Parameters

dictionary
[topologic_core.Dictionary or dict] The input dictionary.

key
[string] The input key.

Returns

topologic_core.Dictionary or dict
The input dictionary with the key/value removed from it.

`static SetValueAtKey`(*dictionary*, *key*, *value*)

Creates a key/value pair in the input dictionary.

Parameters

dictionary
[topologic_core.Dictionary or dict] The input dictionary.

key
[string] The input key.

value
[int , float , string, or list] The value associated with the key.

Returns

topologic_core.Dictionary or dict
The input dictionary with the key/value pair added to it.

`static ValueAtKey`(*dictionary*, *key*, *silent=False*)

Returns the value of the input key in the input dictionary.

Parameters

dictionary
[topologic_core.Dictionary or dict] The input dictionary.

key
[string] The input key.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

int , float, string, list , or dict
The value found at the input key in the input dictionary.

static Values(*dictionary*)

Returns the list of values in the input dictionary.

Parameters

dictionary
[topologic_core.Dictionary or dict] The input dictionary.

Returns

list
The list of values found in the input dictionary.

topologicpy.Edge module

class topologicpy.Edge.**Edge**

Bases: object

Methods

Angle(edgeA, edgeB[, mantissa, bracket])	Returns the angle in degrees between the two input edges.
Bisect(edgeA, edgeB[, length, placement, ...])	Creates a bisecting edge between edgeA and edgeB.
ByFaceNormal(face[, origin, length, tolerance])	Creates a straight edge representing the normal to the input face.
ByOffset2D(edge[, offset, tolerance])	Creates and edge offset from the input edge.
ByStartVertexEndVertex(vertexA, vertexB[, ...])	Creates a straight edge that connects the input vertices.
ByVertices(*args[, tolerance, silent])	Creates a straight edge that connects the input list of vertices.
ByVerticesCluster(cluster[, tolerance])	Creates a straight edge that connects the input cluster of vertices.
Connection(edgeA, edgeB[, tolerance, silent])	Returns the edge representing the connection between the first input edge to the second input edge using the two closest vertices.
Direction(edge[, mantissa])	Returns the direction of the input edge expressed as a list of three numbers.
EndVertex(edge)	Returns the end vertex of the input edge.
Equation2D(edge[, mantissa])	Returns the 2D equation of the input edge.
Extend(edge[, distance, bothSides, reverse, ...])	Extends the input edge by the input distance.
ExtendToEdge(edgeA, edgeB[, mantissa, step, ...])	Extends the first input edge to meet the second input edge.
ExternalBoundary(edge)	Returns the external boundary (cluster of end vertices) of the input edge.
Index(edge, edges[, strict, tolerance])	Returns index of the input edge in the input list of edges
Intersect2D(edgeA, edgeB[, silent, ...])	Returns the intersection vertex of the two input edges.
IsCollinear(edgeA, edgeB[, mantissa, tolerance])	Return True if the two input edges are collinear.
IsCoplanar(edgeA, edgeB[, mantissa, tolerance])	Return True if the two input edges are coplanar.
IsParallel(edgeA, edgeB[, mantissa, tolerance])	Return True if the two input edges are parallel.
Length(edge[, mantissa])	Returns the length of the input edge.

continues on next page

Table 2 – continued from previous page

Line([origin, length, direction, placement, ...])	Creates a straight edge (line) using the input parameters.
Normal(edge[, angle])	Returns the normal (perpendicular) vector to the input edge.
NormalEdge(edge[, length, u, angle, ...])	Returns the normal (perpendicular) vector to the input edge as an edge.
Normalize(edge[, useEndVertex, tolerance])	Creates a normalized edge that has the same direction as the input edge, but a length of 1.
ParameterAtVertex(edge, vertex[, mantissa, ...])	Returns the *u* parameter along the input edge based on the location of the input vertex.
Reverse(edge[, tolerance])	Creates an edge that has the reverse direction of the input edge.
SetLength(edge[, length, bothSides, ...])	Returns an edge with the new length in the same direction as the input edge.
StartVertex(edge)	Returns the start vertex of the input edge.
Trim(edge[, distance, bothSides, reverse, ...])	Trims the input edge by the input distance.
TrimByEdge(edgeA, edgeB[, reverse, ...])	Trims the first input edge by the second input edge.
VertexByDistance(edge[, distance, origin, ...])	Creates a vertex along the input edge offset by the input distance from the input origin.
VertexByParameter(edge[, u])	Creates a vertex along the input edge offset by the input *u* parameter.
Vertices(edge)	Returns the list of vertices of the input edge.

static Angle(*edgeA*, *edgeB*, *mantissa: int = 6*, *bracket: bool = False*) → float

Returns the angle in degrees between the two input edges.

Parameters

edgeA
[topologic_core.Edge] The first input edge.

edgeB
[topologic Edge] The second input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

bracket
[bool] If set to True, the returned angle is bracketed between 0 and 180. The default is False.

Returns

float
The angle in degrees between the two input edges.

static Bisect(*edgeA*, *edgeB*, *length: float = 1.0*, *placement: int = 0*, *tolerance: float = 0.0001*)

Creates a bisecting edge between edgeA and edgeB.

Parameters

edgeA
[topologic_core.Edge] The first topologic Edge.

edgeB
[topologic Edge] The second topologic Edge.

length
[float , optional] The desired length of the bisecting edge. The default is 1.0.

placement
[int , optional] The desired placement of the bisecting edge. If set to 0, the bisecting edge centroid will be placed at the end vertex of the first edge. If set to 1, the bisecting edge start vertex will be placed at the end vertex of the first edge. If set to 2, the bisecting edge end vertex will be placed at the end vertex of the first edge. If set to any number other than 0, 1, or 2, the bisecting edge centroid will be placed at the end vertex of the first edge. The default is 0.

tolerance
[float , optional] The desired tolerance to decide if an Edge can be created. The default is 0.0001.

Returns

topologic_core.Edge
The created bisecting edge.

`static ByFaceNormal`(*face*, *origin=None*, *length: float = 1.0*, *tolerance: float = 0.0001*)

Creates a straight edge representing the normal to the input face.

Parameters

face
[topologic_core.Face] The input face

origin
[topologic_core.Vertex , optional] The desired origin of the edge. If set to None, the centroid of the face is chosen as the origin of the edge. The default is None.

length
[float , optional] The desired length of the edge. The default is 1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

edge
[topologic_core.Edge] The created edge.

`static ByOffset2D`(*edge*, *offset: float = 1.0*, *tolerance: float = 0.0001*)

Creates and edge offset from the input edge. This method is intended for edges that are in the XY plane.

Parameters

edge
[topologic_core.Edge] The input edge.

offset
[float , optional] The desired offset. The default is 1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
An edge offset from the input edge.

`static ByStartVertexEndVertex`(*vertexA*, *vertexB*, *tolerance: float = 0.0001*, *silent=False*)

Creates a straight edge that connects the input vertices.

Parameters

vertexA
[topologic_core.Vertex] The first input vertex. This is considered the start vertex.

vertexB
[topologic_core.Vertex] The second input vertex. This is considered the end vertex.

tolerance
[float , optional] The desired tolerance to decide if an Edge can be created. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

edge
[topologic_core.Edge] The created edge.

`static ByVertices`(**args*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a straight edge that connects the input list of vertices.

Parameters

vertices
[list] The input list of vertices. The first item is considered the start vertex and the last item is considered the end vertex.

tolerance
[float , optional] The desired tolerance to decide if an edge can be created. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Edge
The created edge.

`static ByVerticesCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a straight edge that connects the input cluster of vertices.

Parameters

cluster
[topologic_core.Cluster] The input cluster of vertices. The first item is considered the start vertex and the last item is considered the end vertex.

tolerance
[float , optional] The desired tolerance to decide if an edge can be created. The default is 0.0001.

Returns

topologic_core.Edge
The created edge.

`static Connection`(*edgeA*, *edgeB*, *tolerance: float = 0.0001*, *silent: bool = False*)

Returns the edge representing the connection between the first input edge to the second input edge using the two closest vertices.

Parameters

edgeA
[topologic_core.Edge] The first input edge. This edge will be extended to meet edgeB.

edgeB
[topologic_core.Edge] The second input edge. This edge will be used to extend edgeA.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge or topologic_core.Wire
The connected edge. Since it is made of two edges, this method returns a Wire.

`static Direction`(*edge*, *mantissa: int = 6*) → list

Returns the direction of the input edge expressed as a list of three numbers.

Parameters

edge
[topologic_core.Edge] The input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The direction of the input edge.

`static EndVertex`(*edge*)

Returns the end vertex of the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

Returns

topologic_core.Vertex
The end vertex of the input edge.

`static Equation2D`(*edge*, *mantissa=6*)

Returns the 2D equation of the input edge. This is assumed to be in the XY plane.

Parameters

edge
[topologic_core.Edge] The input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
The equation of the edge stored in a dictionary. The dictionary has the following keys: "slope": The slope of the line. This can be float('inf') "x_intercept": The X axis intercept. This can be None. "y_intercept": The Y axis intercept. This can be None.

`static Extend`(*edge*, *distance: float = 1.0*, *bothSides: bool = True*, *reverse: bool = False*, *tolerance: float = 0.0001*)

Extends the input edge by the input distance.

Parameters

edge
[topologic_core.Edge] The input edge.

distance
[float , optional] The offset distance. The default is 1.

bothSides
[bool , optional] If set to True, the edge will be extended by half the distance at each end. The default is False.

reverse
[bool , optional] If set to True, the edge will be extended from its start vertex. Otherwise, it will be extended from its end vertex. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The extended edge.

`static ExtendToEdge`(*edgeA*, *edgeB*, *mantissa: int = 6*, *step: bool = True*, *tolerance: float = 0.0001*, *silent: bool = False*)

Extends the first input edge to meet the second input edge.

Parameters

edgeA
[topologic_core.Edge] The first input edge. This edge will be extended to meet edgeB.

edgeB
[topologic_core.Edge] The second input edge. This edge will be used to extend edgeA.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Edge
The extended edge.

static ExternalBoundary(*edge*)

Returns the external boundary (cluster of end vertices) of the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

Returns

topologic_core.Cluster
The external boundary of the input edge. This is a cluster of the edge's end vertices.

static Index(*edge*, *edges: list*, *strict: bool = False*, *tolerance: float = 0.0001*) → int

Returns index of the input edge in the input list of edges

Parameters

edge
[topologic_core.Edge] The input edge.

edges
[list] The input list of edges.

strict
[bool , optional] If set to True, the edge must be strictly identical to the one found in the list. Otherwise, a distance comparison is used. The default is False.

tolerance
[float , optional] The tolerance for computing if the input edge is identical to an edge from the list. The default is 0.0001.

Returns

int
The index of the input edge in the input list of edges.

static Intersect2D(*edgeA*, *edgeB*, *silent: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the intersection vertex of the two input edges. This is assumed to be in the XY plane. The intersection vertex does not necessarily fall within the extents of either edge.

Parameters

edgeA
[topologic_core.Edge] The first input edge.

edgeB
[topologic_core.Edge] The second input edge.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic_core.Vertex
The intersection vertex or None if the edges are parallel or collinear.

`static IsCollinear`(*edgeA*, *edgeB*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → bool

Return True if the two input edges are collinear. Returns False otherwise. This code is based on a contribution by https://github.com/gaoxipeng

Parameters

edgeA
[topologic_core.Edge] The first input edge.

edgeB
[topologic_core.Edge] The second input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the two edges are collinear. False otherwise.

`static IsCoplanar`(*edgeA*, *edgeB*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Return True if the two input edges are coplanar. Returns False otherwise.

Parameters

edgeA
[topologic_core.Edge] The first input edge.

edgeB
[topologic_core.Edge] The second input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the two edges are coplanar. False otherwise.

`static IsParallel`(*edgeA*, *edgeB*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → bool

Return True if the two input edges are parallel. Returns False otherwise.

Parameters

edgeA
[topologic_core.Edge] The first input edge.

edgeB
[topologic_core.Edge] The second input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the two edges are collinear. False otherwise.

`static Length`(*edge*, *mantissa: int = 6*) → float

Returns the length of the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The length of the input edge.

`static Line`(*origin=None*, *length: float = 1*, *direction: list = [1, 0, 0]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a straight edge (line) using the input parameters.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the box. The default is None which results in the edge being placed at (0, 0, 0).

length
[float , optional] The desired length of the edge. The default is 1.0.

direction
[list , optional] The desired direction (vector) of the edge. The default is [1,0,0] (along the X-axis).

placement
[str , optional] The desired placement of the edge. The options are: 1. "center" which places the center of the edge at the origin. 2. "start" which places the start of the edge at the origin. 3. "end" which places the end of the edge at the origin. The default is "center". It is case insensitive.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

——-

topologic_core.Edge
The created edge

`static Normal`(*edge*, *angle: float = 0.0*)

Returns the normal (perpendicular) vector to the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

angle
[float , optional] The desired rotational offset angle in degrees for the normal edge. This rotates the normal edge by the angle value around the axis defined by the input edge. The default is 0.0.

Returns

list
The normal (perpendicular) vector to the input edge.

`static NormalEdge`(*edge*, *length: float = 1.0*, *u: float = 0.5*, *angle: float = 0.0*, *tolerance: float = 0.0001*, *silent: bool = False*)

Returns the normal (perpendicular) vector to the input edge as an edge.

Parameters

edge
[topologic_core.Edge] The input edge.

length
[float , optional] The desired length of the normal edge. The default is 1.0.

u
[float , optional] The desired u parameter placement of the normal edge. A value of 0.0 places the normal edge at the start vertex of the input edge, a value of 0.5 places the normal edge at the midpoint of the input edge, and a value of 1.0 places the normal edge at the end vertex of the input edge. The default is 0.5

angle
[float , optional] The desired rotational offset angle in degrees for the normal edge. This rotates the normal edge by the angle value around the axis defined by the input edge. The default is 0.0.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Edge
The normal (perpendicular) vector to the input edge as an edge.

`static Normalize`(*edge*, *useEndVertex: bool = False*, *tolerance: float = 0.0001*)

Creates a normalized edge that has the same direction as the input edge, but a length of 1.

Parameters

edge
[topologic_core.Edge] The input edge.

useEndVertex
[bool , optional] If True the normalized edge end vertex will be placed at the end vertex of the input edge. Otherwise, the normalized edge start vertex will be placed at the start vertex of the input edge. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The normalized edge.

static ParameterAtVertex(*edge*, *vertex*, *mantissa: int = 6*, *silent: bool = False*) → float

Returns the *u* parameter along the input edge based on the location of the input vertex.

Parameters

edge
[topologic_core.Edge] The input edge.

vertex
[topologic_core.Vertex] The input vertex.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

float
The *u* parameter along the input edge based on the location of the input vertex.

static Reverse(*edge*, *tolerance: float = 0.0001*)

Creates an edge that has the reverse direction of the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The reversed edge.

static SetLength(*edge*, *length: float = 1.0*, *bothSides: bool = True*, *reverse: bool = False*, *tolerance: float = 0.0001*)

Returns an edge with the new length in the same direction as the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

length
[float , optional] The desired length of the edge. The default is 1.

bothSides
[bool , optional] If set to True, the edge will be offset symmetrically from each end. The default is True.

reverse
[bool , optional] If set to True, the edge will be offset from its start vertex. Otherwise, it will be offset from its end vertex. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The extended edge.

`static StartVertex`(*edge*)

Returns the start vertex of the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

Returns

topologic_core.Vertex
The start vertex of the input edge.

`static Trim`(*edge*, *distance: float = 0.0*, *bothSides: bool = True*, *reverse: bool = False*, *tolerance: float = 0.0001*)

Trims the input edge by the input distance.

Parameters

edge
[topologic_core.Edge] The input edge.

distance
[float , optional] The offset distance. The default is 0.

bothSides
[bool , optional] If set to True, the edge will be trimmed by half the distance at each end. The default is False.

reverse
[bool , optional] If set to True, the edge will be trimmed from its start vertex. Otherwise, it will be trimmed from its end vertex. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The trimmed edge.

`static TrimByEdge`(*edgeA*, *edgeB*, *reverse: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*, *silent: bool = False*)

Trims the first input edge by the second input edge.

Parameters

edgeA
[topologic_core.Edge] The first input edge. This edge will be trimmed by edgeB.

edgeB
[topologic_core.Edge] The second input edge. This edge will be used to trim edgeA.

reverse
[bool , optional] If set to True, which segment is preserved is reversed. Otherwise, it is not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Edge
The trimmed edge.

`static VertexByDistance`(*edge*, *distance: float = 0.0*, *origin=None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a vertex along the input edge offset by the input distance from the input origin.

Parameters

edge
[topologic_core.Edge] The input edge.

distance
[float , optional] The offset distance. The default is 0.

origin
[topologic_core.Vertex , optional] The origin of the offset distance. If set to None, the origin will be set to the start vertex of the input edge. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Vertex
The created vertex.

`static VertexByParameter`(*edge*, *u: float = 0.0*)

Creates a vertex along the input edge offset by the input *u* parameter.

Parameters

edge
[topologic_core.Edge] The input edge.

u
[float , optional] The *u* parameter along the input topologic Edge. A parameter of 0 returns the start vertex. A parameter of 1 returns the end vertex. The default is 0.

Returns

topologic_core.Vertex
The created vertex.

`static Vertices`(*edge*) → list

Returns the list of vertices of the input edge.

Parameters

edge
[topologic_core.Edge] The input edge.

Returns

list
The list of vertices.

topologicpy.EnergyModel module

class topologicpy.EnergyModel.**EnergyModel**

Bases: object

@staticmethod def ByOSMFile(file):

“”” Creates an EnergyModel from the input OSM file path.

Parameters

path
[string] The path to the input .OSM file.

Returns

openstudio.openstudiomodelcore.Model
The OSM model.

“”” **if not file:**
print(“EnergyModel.ByOSMFile - Error: The input path is not valid. Returning None.”)
return None

osModel = file.read()
if osModel.isNull():
print(“EnergyModel.ByOSMFile - Error: The openstudio model is null. Returning None.”)
return None

else:
osModel = osModel.get()

return osModel

Methods

ByOSMPath(path)	Creates an EnergyModel from the input OSM file path.
ByTopology(building[, shadingSurfaces, ...])	Creates an EnergyModel from the input topology and parameters.
ColumnNames(model, reportName, tableName)	Returns the list of column names given an OSM model, report name, and table name.
DefaultConstructionSets(model)	Returns the default construction sets in the input OSM model.
DefaultScheduleSets(model)	Returns the default schedule sets found in the input OSM model.
ExportToGBXML(model, path[, overwrite])	Exports the input OSM model to a GBXML file.
ExportToOSM(model, path[, overwrite])	Exports the input OSM model to an OSM file.
GBXMLString(model)	Returns the GBXML string of the input OSM model.
Query(model[, reportName, reportForString, ...])	Queries the model for values.
ReportNames(model)	Returns the report names found in the input OSM model.
RowNames(model, reportName, tableName)	Returns the list of row names given an OSM model, report name, and table name.
Run(model[, weatherFilePath, osBinaryPath, ...])	Runs an energy simulation.
SpaceColors(model)	Return the space colors found in the input OSM model.
SpaceDictionaries(model)	Return the space dictionaries found in the input OSM model.
SpaceTypeNames(model)	Return the space type names found in the input OSM model.
SpaceTypes(model)	Return the space types found in the input OSM model.
SqlFile(model)	Returns the SQL file found in the input OSM model.
TableNames(model, reportName)	Returns the table names found in the input OSM model and report name.
Topologies(model[, tolerance])	**Parameters**
Units(model, reportName, tableName, columnName)	**Parameters**

static ByOSMPath(*path: str*)

Creates an EnergyModel from the input OSM file path.

Parameters

path
[string] The path to the input .OSM file.

Returns

openstudio.openstudiomodelcore.Model
The OSM model.

static ByTopology(*building*, *shadingSurfaces=None*, *osModelPath: str = None*, *weatherFilePath: str = None*, *designDayFilePath: str = None*, *floorLevels: list = None*, *buildingName: str = 'TopologicBuilding'*, *buildingType: str = 'Commercial'*, *northAxis: float = 0.0*, *glazingRatio: float = 0.0*, *coolingTemp: float = 25.0*, *heatingTemp: float = 20.0*, *defaultSpaceType: str = '189.1-2009 - Office - WholeBuilding - Lg Office - CZ4-8'*, *spaceNameKey: str = 'TOPOLOGIC_name'*, *spaceTypeKey: str = 'TOPOLOGIC_type'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates an EnergyModel from the input topology and parameters.

Parameters

building
[topologic_core.CellComplex or topologic_core.Cell] The input building topology.

shadingSurfaces
[topologic_core.Topology , optional] The input topology for shading surfaces. The default is None.

osModelPath
[str , optional] The path to the template OSM file. The default is "./assets/EnergyModel/OSMTemplate-OfficeBuilding-3.5.0.osm".

weatherFilePath
[str , optional] The input energy plus weather (epw) file. The default is "./assets/EnergyModel/GBR_London.Gatwick.037760_IWEC.epw".

designDayFilePath
[str , optional] The input design day (ddy) file path. The default is "./assets/EnergyModel/GBR_London.Gatwick.037760_IWEC.ddy",

floorLevels
[list , optional] The list of floor level Z heights including the lowest most and the highest most levels. If set to None, this method will attempt to find the floor levels from the horizontal faces of the input topology

buildingName
[str , optional] The desired name of the building. The default is "TopologicBuilding".

buildingType
[str , optional] The building type. The default is "Commercial".

defaultSpaceType
[str , optional] The default space type to apply to spaces that do not have a type assigned in their dictionary. The default is "189.1-2009 - Office - WholeBuilding - Lg Office - CZ4-8".

northAxis
[float , optional] The counter-clockwise angle in degrees from the positive Y-axis representing the direction of the north axis. The default is 0.0.

glazingRatio
[float , optional] The glazing ratio (ratio of windows to wall) to use for exterior vertical walls that do not have apertures. If you do not wish to use a glazing ratio, set it to 0. The default is 0.

coolingTemp
[float , optional] The desired temperature in degrees at which the cooling system should activate. The default is 25.0.

heatingTemp
[float , optional] The desired temperature in degrees at which the heating system should activate. The default is 25.0..

spaceNameKey
[str , optional] The dictionary key to use to find the space name value. The default is "Name".

spaceTypeKey
[str , optional] The dictionary key to use to find the space type value. The default is "Type".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

openstudio.openstudiomodelcore.Model
The created OSM model.

`static ColumnNames`(*model*, *reportName*, *tableName*)

Returns the list of column names given an OSM model, report name, and table name.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

reportName
[str] The input report name.

tableName
[str] The input table name.

Returns

list
the list of column names.

`static DefaultConstructionSets`(*model*)

Returns the default construction sets in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

list
The default construction sets.

`static DefaultScheduleSets`(*model*)

Returns the default schedule sets found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

list
The list of default schedule sets.

`static ExportToGBXML(model, path, overwrite=False)`

Exports the input OSM model to a GBXML file.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

path
[str] The path for saving the file.

overwrite
[bool, optional] If set to True any file with the same name is over-written. The default is False.

Returns

bool
True if the file is written successfully. False otherwise.

`static ExportToOSM(model, path, overwrite=False)`

Exports the input OSM model to an OSM file.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

path
[str] The path for saving the file.

overwrite
[bool, optional] If set to True any file with the same name is over-written. The default is False.

Returns

bool
True if the file is written successfully. False otherwise.

`static GBXMLString(model)`

Returns the GBXML string of the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

str
The gbxml string.

`static Query`(*model*, *reportName: str = 'HVACSizingSummary'*, *reportForString: str = 'Entire Facility'*, *tableName: str = 'Zone Sensible Cooling'*, *columnName: str = 'Calculated Design Load'*, *rowNames: list = []*, *units: str = 'W'*)

Queries the model for values.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

reportName
[str , optional] The input report name. The default is "HVACSizingSummary".

reportForString
[str, optional] The input report for string. The default is "Entire Facility".

tableName
[str , optional] The input table name. The default is "Zone Sensible Cooling".

columnName
[str , optional] The input column name. The default is "Calculated Design Load".

rowNames
[list , optional] The input list of row names. The default is [].

units
[str , optional] The input units. The default is "W".

Returns

list
The list of values.

`static ReportNames`(*model*)

Returns the report names found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

list
The list of report names found in the input OSM model.

`static RowNames`(*model*, *reportName*, *tableName*)

Returns the list of row names given an OSM model, report name, and table name.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

reportName
[str] The input name of the report.

tableName
[str] The input name of the table.

Returns

list
The list of row names.

`static Run`(*model*, *weatherFilePath: str = None*, *osBinaryPath: str = None*, *outputFolder: str = None*, *removeFiles: bool = False*)

Runs an energy simulation.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

weatherFilePath
[str] The path to the epw weather file.

osBinaryPath
[str] The path to the openstudio binary.

outputFolder
[str] The path to the output folder.

removeFiles
[bool , optional] If set to True, the working files are removed at the end of the process. The default is False.

Returns

model
[openstudio.openstudiomodelcore.Model] The simulated OSM model.

`static SpaceColors`(*model*)

Return the space colors found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

list
The list of space colors. Each item is a three-item list representing the red, green, and blue values of the color.

`static SpaceDictionaries`(*model*)

Return the space dictionaries found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

dict
The dictionary of space types, names, and colors found in the input OSM model. The dictionary has the following keys: - "types" - "names" - "colors"

`static SpaceTypeNames(`*model*`)`

Return the space type names found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

list
The list of space type names

`static SpaceTypes(`*model*`)`

Return the space types found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

list
The list of space types

`static SqlFile(`*model*`)`

Returns the SQL file found in the input OSM model.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

Returns

SQL file
The SQL file found in the input OSM model.

`static TableNames(`*model*, *reportName*`)`

Returns the table names found in the input OSM model and report name.

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

reportName
[str] The input report name.

Returns

list
The list of table names found in the input OSM model and report name.

static **Topologies**(*model*, *tolerance=0.0001*)

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dict
The dictionary of topologies found in the input OSM model. The keys of the dictionary are: - "cells" - "apertures" - "shadingFaces"

static **Units**(*model*, *reportName*, *tableName*, *columnName*)

Parameters

model
[openstudio.openstudiomodelcore.Model] The input OSM model.

reportName
[str] The input report name.

tableName
[str] The input table name.

columnName
[str] The input column name.

Returns

str
The units string found in the input OSM model, report name, table name, and column name.

topologicpy.Face module

class topologicpy.Face.**Face**

Bases: object

Methods

AddInternalBoundaries(face, wires)	Adds internal boundaries (closed wires) to the input face.
AddInternalBoundariesCluster(face, cluster)	Adds the input cluster of internal boundaries (closed wires) to the input face.
Angle(faceA, faceB[, mantissa])	Returns the angle in degrees between the two input faces.
Area(face[, mantissa])	Returns the area of the input face.
BoundingRectangle(topology[, optimize, ...])	Returns a face representing a bounding rectangle of the input topology.
ByEdges(edges[, tolerance])	Creates a face from the input list of edges.
ByEdgesCluster(cluster[, tolerance])	Creates a face from the input cluster of edges.
ByOffset(face[, offset, offsetKey, ...])	Creates an offset face from the input face.

continues on next page

Table 3 – continued from previous page

ByOffsetArea(face, area[, offsetKey, ...])	Creates an offset face from the input face based on the input area.
ByShell(shell[, origin, angTolerance, ...])	Creates a face by merging the faces of the input shell.
ByThickenedWire(wire[, offsetA, offsetB, ...])	Creates a face by thickening the input wire.
ByVertices(vertices[, tolerance])	Creates a face from the input list of vertices.
ByVerticesCluster(cluster[, tolerance])	Creates a face from the input cluster of vertices.
ByWire(wire[, tolerance, silent])	Creates a face from the input closed wire.
ByWires(externalBoundary[, ...])	Creates a face from the input external boundary (closed wire) and the input list of internal boundaries (closed wires).
ByWiresCluster(externalBoundary[, ...])	Creates a face from the input external boundary (closed wire) and the input cluster of internal boundaries (closed wires).
Circle([origin, radius, sides, fromAngle, ...])	Creates a circle.
Compactness(face[, mantissa])	Returns the compactness measure of the input face.
CompassAngle(face[, north, mantissa, tolerance])	Returns the horizontal compass angle in degrees between the normal vector of the input face and the input vector.
Edges(face)	Returns the edges of the input face.
Einstein([origin, radius, direction, ...])	Creates an aperiodic monotile, also called an 'einstein' tile (meaning one tile in German, not the name of the famous physicist).
Ellipse([origin, inputMode, width, length, ...])	Creates an ellipse and returns all its geometry and parameters.
ExteriorAngles(face[, ...])	Returns the exterior angles of the input face in degrees.
ExternalBoundary(face[, silent])	Returns the external boundary (closed wire) of the input face.
FacingToward(face[, direction, asVertex, ...])	Returns True if the input face is facing toward the input direction.
Fillet(face[, radius, radiusKey, tolerance, ...])	Fillets (rounds) the interior and exterior corners of the input face given the input radius.
Harmonize(face[, tolerance])	Returns a harmonized version of the input face such that the u and v origins are always in the upperleft corner.
InteriorAngles(face[, ...])	Returns the interior angles of the input face in degrees.
InternalBoundaries(face)	Returns the internal boundaries (closed wires) of the input face.
InternalVertex(face[, tolerance])	Creates a vertex guaranteed to be inside the input face.
Invert(face[, tolerance])	Creates a face that is an inverse (mirror) of the input face.
IsCoplanar(faceA, faceB[, mantissa, tolerance])	Returns True if the two input faces are coplanar.
Isovist(face, vertex[, obstacles, ...])	Returns the face representing the isovist projection from the input viewpoint.
MedialAxis(face[, resolution, ...])	Returns a wire representing an approximation of the medial axis of the input topology.
Normal(face[, outputType, mantissa])	Returns the normal vector to the input face.
NormalEdge(face[, length, tolerance, silent])	Returns the normal vector to the input face as an edge with the desired input length.

continues on next page

Table 3 – continued from previous page

NorthArrow([origin, radius, sides, ...])	Creates a north arrow.
Planarize(face[, origin, tolerance])	Planarizes the input face such that its center of mass is located at the input origin and its normal is pointed in the input direction.
PlaneEquation(face[, mantissa])	Returns the a, b, c, d coefficients of the plane equation of the input face.
Project(faceA, faceB[, direction, mantissa, ...])	Creates a projection of the first input face unto the second input face.
Rectangle([origin, width, length, ...])	Creates a rectangle.
RemoveCollinearEdges(face[, angTolerance, ...])	Removes any collinear edges in the input face.
Simplify(face[, tolerance])	Simplifies the input face edges based on the Douglas Peucker algorthim.
Skeleton(face[, boundary, tolerance])	Creates a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com>
Square([origin, size, direction, placement, ...])	Creates a square.
Squircle([origin, radius, sides, a, b, ...])	Creates a Squircle which is a hybrid between a circle and a square.
Star([origin, radiusA, radiusB, rays, ...])	Creates a star.
Trapezoid([origin, widthA, widthB, offsetA, ...])	Creates a trapezoid.
Triangulate(face[, mode, meshSize, ...])	Triangulates the input face and returns a list of faces.
TrimByWire(face, wire[, reverse])	Trims the input face by the input wire.
VertexByParameters(face[, u, v])	Creates a vertex at the *u* and *v* parameters of the input face.
VertexParameters(face, vertex[, outputType, ...])	Returns the *u* and *v* parameters of the input face at the location of the input vertex.
Vertices(face)	Returns the vertices of the input face.
Wire(face)	Returns the external boundary (closed wire) of the input face.
Wires(face)	Returns the wires of the input face.

RectangleByPlaneEquation

`static AddInternalBoundaries`(*face*, *wires: list*)

Adds internal boundaries (closed wires) to the input face. Internal boundaries are considered holes in the input face.

Parameters

face
[topologic_core.Face] The input face.

wires
[list] The input list of internal boundaries (closed wires).

Returns

topologic_core.Face
The created face with internal boundaries added to it.

`static AddInternalBoundariesCluster`(*face*, *cluster*)

Adds the input cluster of internal boundaries (closed wires) to the input face. Internal boundaries are considered holes in the input face.

Parameters

face
[topologic_core.Face] The input face.

cluster
[topologic_core.Cluster] The input cluster of internal boundaries (topologic wires).

Returns

topologic_core.Face
The created face with internal boundaries added to it.

`static Angle`(*faceA*, *faceB*, *mantissa: int = 6*) → float

Returns the angle in degrees between the two input faces.

Parameters

faceA
[topologic_core.Face] The first input face.

faceB
[topologic_core.Face] The second input face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The angle in degrees between the two input faces.

`static Area`(*face*, *mantissa: int = 6*) → float

Returns the area of the input face.

Parameters

face
[topologic_core.Face] The input face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The area of the input face.

`static BoundingRectangle`(*topology*, *optimize: int = 0*, *tolerance: float = 0.0001*)

Returns a face representing a bounding rectangle of the input topology. The returned face contains a dictionary with key "zrot" that represents rotations around the Z axis. If applied the resulting face will become axis-aligned.

Parameters

topology
[topologic_core.Topology] The input topology.

optimize
[int , optional] If set to an integer from 1 (low optimization) to 10 (high optimization), the method will attempt to optimize the bounding rectangle so that it reduces its surface area. The default is 0 which will result in an axis-aligned bounding rectangle. The default is 0.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The bounding rectangle of the input topology.

`static ByEdges`(*edges: list*, *tolerance: float = 0.0001*)

Creates a face from the input list of edges.

Parameters

edges
[list] The input list of edges.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

face
[topologic_core.Face] The created face.

`static ByEdgesCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a face from the input cluster of edges.

Parameters

cluster
[topologic_core.Cluster] The input cluster of edges.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

face
[topologic_core.Face] The created face.

`static ByOffset`(*face*, *offset: float = 1.0*, *offsetKey: str = 'offset'*, *stepOffsetA: float = 0*, *stepOffsetB: float = 0*, *stepOffsetKeyA: str = 'stepOffsetA'*, *stepOffsetKeyB: str = 'stepOffsetB'*, *reverse: bool = False*, *bisectors: bool = False*, *transferDictionaries: bool = False*, *epsilon: float = 0.01*, *tolerance: float = 0.0001*, *silent: bool = False*, *numWorkers: int = None*)

Creates an offset face from the input face. A positive offset value results in an offset to the interior of an anti-clockwise face.

Parameters

face
[topologic_core.Face] The input face.

offset
[float , optional] The desired offset distance. The default is 1.0.

offsetKey
[str , optional] The edge dictionary key under which to find the offset value. If a value cannot be found, the offset input parameter value is used instead. The default is "offset".

stepOffsetA
[float , optional] The amount to offset along the previous edge when transitioning between parallel edges with different offsets. The default is 0.

stepOffsetB
[float , optional] The amount to offset along the next edge when transitioning between parallel edges with different offsets. The default is 0.

stepOffsetKeyA
[str , optional] The vertex dictionary key under which to find the step offset A value. If a value cannot be found, the stepOffsetA input parameter value is used instead. The default is "stepOffsetA".

stepOffsetKeyB
[str , optional] The vertex dictionary key under which to find the step offset B value. If a value cannot be found, the stepOffsetB input parameter value is used instead. The default is "stepOffsetB".

bisectors
[bool , optional] If set to True, The bisectors (seams) edges will be included in the returned wire. This will result in the returned shape to be a shell rather than a face. The default is False.

reverse
[bool , optional] If set to True, the direction of offsets is reversed. Otherwise, it is not. The default is False.

transferDictionaries
[bool , optional] If set to True, the dictionaries of the original wire, its edges, and its vertices are transfered to the new wire. Otherwise, they are not. The default is False.

epsilon
[float , optional] The desired epsilon (another form of tolerance for shortest edge to remove). The default is 0.01. (This is set to a larger number as it was found to work better)

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

numWorkers
[int , optional] Number of workers run in parallel to process. If you set it to 1, no parallel processing will take place. The default is None which causes the algorithm to use twice the number of cpu cores in the host computer.

Returns

topologic_core.Face or topologic_core.Shell
The created face or shell.

`static ByOffsetArea`(*face*, *area*, *offsetKey='offset'*, *minOffsetKey='minOffset'*, *maxOffsetKey='maxOffset'*, *defaultMinOffset=0*, *defaultMaxOffset=1*, *maxIterations=1*, *tolerance=0.0001*, *silent=False*, *numWorkers=None*)

Creates an offset face from the input face based on the input area.

Parameters

face
[topologic_core.Face] The input face.

area
[float] The desired area of the created face.

offsetKey
[str , optional] The edge dictionary key under which to store the offset value. The default is "offset".

minOffsetKey
[str , optional] The edge dictionary key under which to find the desired minimum edge offset value. If a value cannot be found, the defaultMinOffset input parameter value is used instead. The default is "minOffset".

maxOffsetKey
[str , optional] The edge dictionary key under which to find the desired maximum edge offset value. If a value cannot be found, the defaultMaxOffset input parameter value is used instead. The default is "maxOffset".

defaultMinOffset
[float , optional] The desired minimum edge offset distance. The default is 0.

defaultMaxOffset
[float , optional] The desired maximum edge offset distance. The default is 1.

maxIterations: int , optional
The desired maximum number of iterations to attempt to converge on a solution. The default is 1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

numWorkers
[int , optional] Number of workers run in parallel to process. If you set it to 1, no parallel processing will take place. The default is None which causes the algorithm to use twice the number of cpu cores in the host computer.

Returns

topologic_core.Face
The created face.

`static ByShell`(*shell*, *origin=None*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*, *silent=False*)

Creates a face by merging the faces of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

`static ByThickenedWire`(*wire*, *offsetA: float = 1.0*, *offsetB: float = 0.0*, *tolerance: float = 0.0001*)

Creates a face by thickening the input wire. This method assumes the wire is manifold and planar.

Parameters

wire
[topologic_core.Wire] The input wire to be thickened.

offsetA
[float , optional] The desired offset to the exterior of the wire. The default is 1.0.

offsetB
[float , optional] The desired offset to the interior of the wire. The default is 0.0.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

`static ByVertices`(*vertices: list*, *tolerance: float = 0.0001*)

Creates a face from the input list of vertices.

Parameters

vertices
[list] The input list of vertices.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

`static ByVerticesCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a face from the input cluster of vertices.

Parameters

cluster
[topologic_core.Cluster] The input cluster of vertices.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

`static ByWire`(*wire*, *tolerance: float = 0.0001*, *silent=False*)

Creates a face from the input closed wire.

Parameters

wire
[topologic_core.Wire] The input wire.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Face or list
The created face. If the wire is non-planar, the method will attempt to triangulate the wire and return a list of faces.

`static ByWires`(*externalBoundary*, *internalBoundaries: list = []*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a face from the input external boundary (closed wire) and the input list of internal boundaries (closed wires).

Parameters

externalBoundary
[topologic_core.Wire] The input external boundary.

internalBoundaries
[list , optional] The input list of internal boundaries (closed wires). The default is an empty list.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Face
The created face.

`static ByWiresCluster`(*externalBoundary*, *internalBoundariesCluster=None*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a face from the input external boundary (closed wire) and the input cluster of internal boundaries (closed wires).

Parameters

externalBoundary topologic_core.Wire
The input external boundary (closed wire).

internalBoundariesCluster
[topologic_core.Cluster] The input cluster of internal boundaries (closed wires). The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Face
The created face.

`static Circle`(*origin=None*, *radius: float = 0.5*, *sides: int = 16*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a circle.

Parameters

origin
[topologic_core.Vertex, optional] The location of the origin of the circle. The default is None which results in the circle being placed at (0, 0, 0).

radius
[float , optional] The radius of the circle. The default is 1.

sides
[int , optional] The number of sides of the circle. The default is 16.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the circle. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the circle. The default is 360.

direction
[list , optional] The vector representing the up direction of the circle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the circle. This can be "center", "lowerleft", "upperleft", "lowerright", or "upperright". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created circle.

`static Compactness`(*face*, *mantissa: int = 6*) → float

Returns the compactness measure of the input face. See https://en.wikipedia.org/wiki/Compactness_measure_of_a_shape

Parameters

face
[topologic_core.Face] The input face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The compactness measure of the input face.

`static CompassAngle`(*face*, *north: list = None*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → float

Returns the horizontal compass angle in degrees between the normal vector of the input face and the input vector. The angle is measured in counter-clockwise fashion. Only the first two elements of the vectors are considered.

Parameters

face
[topologic_core.Face] The input face.

north
[list , optional] The second vector representing the north direction. The default is the positive YAxis ([0,1,0]).

mantissa
[int, optional] The length of the desired mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

float
The horizontal compass angle in degrees between the direction of the face and the second input vector.

`static Edges`(*face*) → list

Returns the edges of the input face.

Parameters

face
[topologic_core.Face] The input face.

Returns

list
The list of edges.

`static Einstein`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an aperiodic monotile, also called an 'einstein' tile (meaning one tile in German, not the name of the famous physicist). See https://arxiv.org/abs/2303.10798

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the tile. The default is None which results in the tiles first vertex being placed at (0, 0, 0).

radius
[float , optional] The radius of the hexagon determining the size of the tile. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the ellipse. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the hexagon determining the location of the tile. This can be "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created Einstein tile.

static Ellipse(*origin=None*, *inputMode: int = 1*, *width: float = 2.0*, *length: float = 1.0*, *focalLength: float = 0.866025*, *eccentricity: float = 0.866025*, *majorAxisLength: float = 1.0*, *minorAxisLength: float = 0.5*, *sides: float = 32*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *close: bool = True*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an ellipse and returns all its geometry and parameters.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the ellipse. The default is None which results in the ellipse being placed at (0, 0, 0).

inputMode
[int , optional] The method by which the ellipse is defined. The default is 1. Based on the inputMode value, only the following inputs will be considered. The options are: 1. Width and Length (considered inputs: width, length) 2. Focal Length and Eccentricity (considered inputs: focalLength, eccentricity) 3. Focal Length and Minor Axis Length (considered inputs: focalLength, minorAxisLength) 4. Major Axis Length and Minor Axis Length (considered input: majorAxisLength, minorAxisLength)

width
[float , optional] The width of the ellipse. The default is 2.0. This is considered if the inputMode is 1.

length
[float , optional] The length of the ellipse. The default is 1.0. This is considered if the inputMode is 1.

focalLength
[float , optional] The focal length of the ellipse. The default is 0.866025. This is considered if the inputMode is 2 or 3.

eccentricity
[float , optional] The eccentricity of the ellipse. The default is 0.866025. This is considered if the inputMode is 2.

majorAxisLength
[float , optional] The length of the major axis of the ellipse. The default is 1.0. This is considered if the inputMode is 4.

minorAxisLength
[float , optional] The length of the minor axis of the ellipse. The default is 0.5. This is considered if the inputMode is 3 or 4.

sides
[int , optional] The number of sides of the ellipse. The default is 32.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the ellipse. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the ellipse. The default is 360.

close
[bool , optional] If set to True, arcs will be closed by connecting the last vertex to the first vertex. Otherwise, they will be left open.

direction
[list , optional] The vector representing the up direction of the ellipse. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the ellipse. This can be "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created ellipse

`static ExteriorAngles`(*face*, *includeInternalBoundaries=False*, *mantissa: int = 6*) → list

Returns the exterior angles of the input face in degrees. The face must be planar.

Parameters

face
[topologic_core.Face] The input face.

includeInternalBoundaries
[bool , optional] If set to True and if the face has internal boundaries (holes), the returned list will be a nested list where the first list is the list of exterior angles of the external boundary and the second list will contain lists of the exterior angles of each of the internal boundaries (holes). For example: [[270,270,270,270], [[270,270,270,270],[300,300,300]]]. If not, the returned list will be a simple list of interior angles of the external boundary. For example: [270,270,270,270]. Please note that that the interior angles of the internal boundaries are considered to be those interior to the original face. Thus, they are exterior to the internal boundary.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

——-

list
The list of exterior angles.

`static ExternalBoundary`(*face*, *silent=False*)

Returns the external boundary (closed wire) of the input face.

Parameters

face
[topologic_core.Face] The input face.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The external boundary of the input face.

`static FacingToward`(*face*, *direction: list = [0, 0, -1]*, *asVertex: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → bool

Returns True if the input face is facing toward the input direction.

Parameters

face
[topologic_core.Face] The input face.

direction
[list , optional] The input direction. The default is [0,0,-1].

asVertex
[bool , optional] If set to True, the direction is treated as an actual vertex in 3D space. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the face is facing toward the direction. False otherwise.

`static Fillet`(*face*, *radius: float = 0*, *radiusKey: str = None*, *tolerance: float = 0.0001*, *silent: bool = False*)

Fillets (rounds) the interior and exterior corners of the input face given the input radius. See https://en.wikipedia.org/wiki/Fillet_(mechanics)

Parameters

face
[topologic_core.Face] The input face.

radius
[float] The desired radius of the fillet.

radiusKey
[str , optional] If specified, the dictionary of the vertices will be queried for this key to specify the desired fillet radius. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Face
The filleted face.

`static Harmonize`(*face*, *tolerance: float = 0.0001*)

Returns a harmonized version of the input face such that the *u* and *v* origins are always in the upperleft corner.

Parameters

face
[topologic_core.Face] The input face.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The harmonized face.

`static InteriorAngles`(*face*, *includeInternalBoundaries: bool = False*, *mantissa: int = 6*) → list

Returns the interior angles of the input face in degrees. The face must be planar.

Parameters

face
[topologic_core.Face] The input face.

includeInternalBoundaries
[bool , optional] If set to True and if the face has internal boundaries (holes), the returned list will be a nested list where the first list is the list of interior angles of the external boundary and the second list will contain lists of the interior angles of each of the internal boundaries (holes). For example: [[90,90,90,90], [[90,90,90,90],[60,60,60]]]. If not, the returned list will be a simple list of interior angles of the external boundary. For example: [90,90,90,90]. Please note that that the interior angles of the internal boundaries are considered to be those interior to the original face. Thus, they are exterior to the internal boundary.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns
——-
list
The list of interior angles.

`static InternalBoundaries`(*face*) → list

Returns the internal boundaries (closed wires) of the input face.

Parameters

face
[topologic_core.Face] The input face.

Returns

list
The list of internal boundaries (closed wires).

`static InternalVertex`(*face*, *tolerance: float = 0.0001*)

Creates a vertex guaranteed to be inside the input face.

Parameters

face
[topologic_core.Face] The input face.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Vertex
The created vertex.

`static Invert`(*face*, *tolerance: float = 0.0001*)

Creates a face that is an inverse (mirror) of the input face.

Parameters

face
[topologic_core.Face] The input face.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The inverted face.

`static IsCoplanar`(*faceA*, *faceB*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → bool

Returns True if the two input faces are coplanar. Returns False otherwise.

Parameters

faceA
[topologic_core.Face] The first input face.

faceB
[topologic_core.Face] The second input face

mantissa
[int , optional] The length of the desired mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the two input faces are coplanar. False otherwise.

`static Isovist`(*face*, *vertex*, *obstacles: list = []*, *direction: list = [0, 1, 0]*, *fov: float = 360*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the face representing the isovist projection from the input viewpoint. This method assumes all input is in 2D. Z coordinates are ignored.

Parameters

face
[topologic_core.Face] The face representing the boundary of the isovist.

vertex
[topologic_core.Vertex] The vertex representing the location of the viewpoint of the isovist.

obstacles
[list , optional] A list of wires representing the obstacles within the face. All obstacles are assumed to be within the boundary of the face. The default is [].

direction
[list, optional] The vector representing the direction (in the XY plane) in which the observer is facing. The Z component is ignored. The direction follows the Vector.CompassAngle convention where [0,1,0] (North) is considered to be in the positive Y direction, [1,0,0] (East) is considered to be in the positive X-direction. Angles are measured in a clockwise fashion. The default is [0,1,0] (North).

fov
[float , optional] The horizontal field of view (fov) angle in degrees. See https://en.wikipedia.org/wiki/Field_of_view. The acceptable range is 1 to 360. The default is 360.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional:] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The face representing the isovist projection from the input viewpoint.

`static MedialAxis`(*face*, *resolution: int = 0*, *externalVertices: bool = False*, *internalVertices: bool = False*, *toLeavesOnly: bool = False*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Returns a wire representing an approximation of the medial axis of the input topology. See https://en.wikipedia.org/wiki/Medial_axis.

Parameters

face
[topologic_core.Face] The input face.

resolution
[int , optional] The desired resolution of the solution (range is 0: standard resolution to 10: high resolution). This determines the density of the sampling along each edge. The default is 0.

externalVertices
[bool , optional] If set to True, the external vertices of the face will be connected to the nearest vertex on the medial axis. The default is False.

internalVertices
[bool , optional] If set to True, the internal vertices of the face will be connected to the nearest vertex on the medial axis. The default is False.

toLeavesOnly
[bool , optional] If set to True, the vertices of the face will be connected to the nearest vertex on the medial axis only if this vertex is a leaf (end point). Otherwise, it will connect to any nearest vertex. The default is False.

angTolerance
[float , optional] The desired angular tolerance in degrees for removing collinear edges. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The medial axis of the input face.

`static Normal`(*face*, *outputType='xyz'*, *mantissa=6*)

Returns the normal vector to the input face. A normal vector of a face is a vector perpendicular to it.

Parameters

face
[topologic_core.Face] The input face.

outputType
[string , optional] The string defining the desired output. This can be any subset or permutation of "xyz". It is case insensitive. The default is "xyz".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The normal vector to the input face.

`static NormalEdge`(*face*, *length: float = 1.0*, *tolerance: float = 0.0001*, *silent: bool = False*)

Returns the normal vector to the input face as an edge with the desired input length. A normal vector of a face is a vector perpendicular to it.

Parameters

face
[topologic_core.Face] The input face.

length
[float , optional] The desired length of the normal edge. The default is 1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The created normal edge to the input face. This is computed at the approximate center of the face.

`static NorthArrow`(*origin=None*, *radius: float = 0.5*, *sides: int = 16*, *direction: list = [0, 0, 1]*, *northAngle: float = 0.0*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a north arrow.

Parameters

origin
[topologic_core.Vertex, optional] The location of the origin of the circle. The default is None which results in the circle being placed at (0, 0, 0).

radius
[float , optional] The radius of the circle. The default is 1.

sides
[int , optional] The number of sides of the circle. The default is 16.

direction
[list , optional] The vector representing the up direction of the circle. The default is [0, 0, 1].

northAngle
[float , optional] The angular offset in degrees from the positive Y axis direction. The angle is measured in a counter-clockwise fashion where 0 is positive Y, 90 is negative X, 180 is negative Y, and 270 is positive X.

placement
[str , optional] The description of the placement of the origin of the circle. This can be “center”, “lowerleft”, “upperleft”, “lowerright”, or “upperright”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created circle.

`static Planarize`(*face*, *origin=None*, *tolerance: float = 0.0001*)

Planarizes the input face such that its center of mass is located at the input origin and its normal is pointed in the input direction.

Parameters

face
[topologic_core.Face] The input face.

origin
[topologic_core.Vertex , optional] The desired vertex to use as the origin of the plane to project the face unto. If set to None, the centroidof the input face is used. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The planarized face.

`static PlaneEquation`(*face*, *mantissa: int = 6*) → dict

Returns the a, b, c, d coefficients of the plane equation of the input face. The input face is assumed to be planar.

Parameters

face
[topologic_core.Face] The input face.

Returns

dict
The dictionary containing the coefficients of the plane equation. The keys of the dictionary are: [“a”, “b”, “c”, “d”].

`static Project`(*faceA*, *faceB*, *direction: list = None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a projection of the first input face unto the second input face.

Parameters

faceA
[topologic_core.Face] The face to be projected.

faceB
[topologic_core.Face] The face unto which the first input face will be projected.

direction
[list, optional] The vector direction of the projection. If None, the reverse vector of the receiving face normal will be used. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The projected Face.

static Rectangle(*origin=None*, *width: float = 1.0*, *length: float = 1.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a rectangle.

Parameters

origin
[topologic_core.Vertex, optional] The location of the origin of the rectangle. The default is None which results in the rectangle being placed at (0, 0, 0).

width
[float , optional] The width of the rectangle. The default is 1.0.

length
[float , optional] The length of the rectangle. The default is 1.0.

direction
[list , optional] The vector representing the up direction of the rectangle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the rectangle. This can be "center", "lowerleft", "upperleft", "lowerright", "upperright". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

static RectangleByPlaneEquation(*origin=None*, *width: float = 1.0*, *length: float = 1.0*, *placement: str = 'center'*, *equation: dict = None*, *tolerance: float = 0.0001*)

static RemoveCollinearEdges(*face*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Removes any collinear edges in the input face.

Parameters

face
[topologic_core.Face] The input face.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face without any collinear edges.

static Simplify(*face*, *tolerance=0.0001*)

Simplifies the input face edges based on the Douglas Peucker algorthim. See https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm Part of this code was contributed by gaoxipeng. See https://github.com/wassimj/topologicpy/issues/35

Parameters

face
[topologic_core.Face] The input face.

tolerance
[float , optional] The desired tolerance. The default is 0.0001. Edges shorter than this length will be removed.

Returns

topologic_core.Face
The simplified face.

`static Skeleton`(*face*, *boundary: bool = True*, *tolerance: float = 0.001*)

Creates a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com> This algorithm depends on the polyskel code which is included in the library. Polyskel code is found at: https://github.com/Botffy/polyskel

Parameters

face
[topologic_core.Face] The input face.

boundary
[bool , optional] If set to True the original boundary is returned as part of the roof. Otherwise it is not. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.001. (This is set to a larger number than the usual 0.0001 as it was found to work better)

Returns

topologic_core.Wire
The created straight skeleton.

`static Square`(*origin=None*, *size: float = 1.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a square.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the square. The default is None which results in the square being placed at (0, 0, 0).

size
[float , optional] The size of the square. The default is 1.0.

direction
[list , optional] The vector representing the up direction of the square. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the square. This can be “center”, “lowerleft”, “upperleft”, “lowerright”, or “upperright”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created square.

`static Squircle`(*origin=None*, *radius: float = 0.5*, *sides: int = 121*, *a: float = 2.0*, *b: float = 2.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Creates a Squircle which is a hybrid between a circle and a square. See https://en.wikipedia.org/wiki/Squircle

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the squircle. The default is None which results in the squircle being placed at (0, 0, 0).

radius
[float , optional] The desired radius of the squircle. The default is 0.5.

sides
[int , optional] The desired number of sides of the squircle. The default is 121.

a
[float , optional] The "a" factor affects the x position of the points to interpolate between a circle and a square. A value of 1 will create a circle. Higher values will create a more square-like shape. The default is 2.0.

b
[float , optional] The "b" factor affects the y position of the points to interpolate between a circle and a square. A value of 1 will create a circle. Higher values will create a more square-like shape. The default is 2.0.

direction
[list , optional] The vector representing the up direction of the circle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the circle. This can be "center", "lowerleft", "upperleft", "lowerright", or "upperright". It is case insensitive. The default is "center".

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created squircle.

`static Star`(*origin=None*, *radiusA: float = 0.5*, *radiusB: float = 0.2*, *rays: int = 8*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a star.

Parameters

origin
[topologic_core.Vertex, optional] The location of the origin of the star. The default is None which results in the star being placed at (0, 0, 0).

radiusA
[float , optional] The outer radius of the star. The default is 1.0.

radiusB
[float , optional] The outer radius of the star. The default is 0.4.

rays
[int , optional] The number of star rays. The default is 5.

direction
[list , optional] The vector representing the up direction of the star. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the star. This can be "center", "lowerleft", "upperleft", "lowerright", or "upperright". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

`static Trapezoid`(*origin=None*, *widthA: float = 1.0*, *widthB: float = 0.75*, *offsetA: float = 0.0*, *offsetB: float = 0.0*, *length: float = 1.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a trapezoid.

Parameters

origin
[topologic_core.Vertex, optional] The location of the origin of the trapezoid. The default is None which results in the trapezoid being placed at (0, 0, 0).

widthA
[float , optional] The width of the bottom edge of the trapezoid. The default is 1.0.

widthB
[float , optional] The width of the top edge of the trapezoid. The default is 0.75.

offsetA
[float , optional] The offset of the bottom edge of the trapezoid. The default is 0.0.

offsetB
[float , optional] The offset of the top edge of the trapezoid. The default is 0.0.

length
[float , optional] The length of the trapezoid. The default is 1.0.

direction
[list , optional] The vector representing the up direction of the trapezoid. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the trapezoid. This can be "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created trapezoid.

static Triangulate(*face*, *mode: int = 0*, *meshSize: float = None*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → list

Triangulates the input face and returns a list of faces.

Parameters

face
[topologic_core.Face] The input face.

mode
[int , optional] The desired mode of meshing algorithm. Several options are available: 0: Classic 1: MeshAdapt 3: Initial Mesh Only 5: Delaunay 6: Frontal-Delaunay 7: BAMG 8: Fontal-Delaunay for Quads 9: Packing of Parallelograms All options other than 0 (Classic) use the gmsh library. See https://gmsh.info/doc/texinfo/gmsh.html#Mesh-options WARNING: The options that use gmsh can be very time consuming and can create very heavy geometry.

meshSize
[float , optional] The desired size of the mesh when using the "mesh" option. If set to None, it will be calculated automatically and set to 10% of the overall size of the face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of triangles of the input face.

static TrimByWire(*face*, *wire*, *reverse: bool = False*)

Trims the input face by the input wire.

Parameters

face
[topologic_core.Face] The input face.

wire
[topologic_core.Wire] The input wire.

reverse
[bool , optional] If set to True, the effect of the trim will be reversed. The default is False.

Returns

topologic_core.Face
The resulting trimmed face.

static VertexByParameters(*face*, *u: float = 0.5*, *v: float = 0.5*)

Creates a vertex at the *u* and *v* parameters of the input face.

Parameters

face
[topologic_core.Face] The input face.

u
[float , optional] The *u* parameter of the input face. The default is 0.5.

v
[float , optional] The *v* parameter of the input face. The default is 0.5.

Returns

vertex
[topologic vertex] The created vertex.

`static VertexParameters`(*face*, *vertex*, *outputType: str = 'uv'*, *mantissa: int = 6*) → list

Returns the *u* and *v* parameters of the input face at the location of the input vertex.

Parameters

face
[topologic_core.Face] The input face.

vertex
[topologic_core.Vertex] The input vertex.

outputType
[string , optional] The string defining the desired output. This can be any subset or permutation of "uv". It is case insensitive. The default is "uv".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The list of *u* and/or *v* as specified by the outputType input.

`static Vertices`(*face*) → list

Returns the vertices of the input face.

Parameters

face
[topologic_core.Face] The input face.

Returns

list
The list of vertices.

`static Wire`(*face*)

Returns the external boundary (closed wire) of the input face.

Parameters

face
[topologic_core.Face] The input face.

Returns

topologic_core.Wire
The external boundary of the input face.

`static Wires`(*face*) → list

Returns the wires of the input face.

Parameters

face
[topologic_core.Face] The input face.

Returns

list
The list of wires.

topologicpy.Graph module

class topologicpy.Graph.**Graph**
Bases: object

Methods

AddEdge(graph, edge[, ...])	Adds the input edge to the input Graph.
AddVertex(graph, vertex[, tolerance, silent])	Adds the input vertex to the input graph.
AddVertices(graph, vertices[, tolerance, silent])	Adds the input vertex to the input graph.
AdjacencyDictionary([vertexLabelKey, ...])	Returns the adjacency dictionary of the input Graph.
AdjacencyList(graph[, vertexKey, reverse, ...])	Returns the adjacency list of the input Graph.
AdjacencyMatrix(graph[, vertexKey, reverse, ...])	Returns the adjacency matrix of the input Graph.
AdjacentVertices(graph, vertex[, silent])	Returns the list of vertices connected to the input vertex.
AllPaths(graph, vertexA, vertexB[, ...])	Returns all the paths that connect the input vertices within the allowed time limit in seconds.
AreIsomorphic(graphA, graphB[, ...])	Tests if the two input graphs are isomorphic according to the Weisfeiler Lehman graph isomorphism test.
AverageClusteringCoefficient(graph[, ...])	Returns the average clustering coefficient of the input graph.
BOTGraph(graph[, bidirectional, ...])	Creates an RDF graph according to the BOT ontology.
BOTString(graph[, format, bidirectional, ...])	Returns an RDF graph serialized string according to the BOT ontology.
BetweenessCentrality(graph[, vertices, ...])	Returns the betweeness centrality measure of the input list of vertices within the input graph. The order of the returned list is the same as the order of the input list of vertices. If no vertices are specified, the betweeness centrality of all the vertices in the input graph is computed. See https://en.wikipedia.org/wiki/Betweenness_centrality.
ByAdjacencyMatrix(adjacencyMatrix[, ...])	Returns graphs according to the input folder path.
ByAdjacencyMatrixCSVPath(path[, ...])	Returns graphs according to the input path.
ByCSVPath(path[, graphIDHeader, ...])	Returns graphs according to the input folder path.
ByDGCNNFile(file[, key, tolerance])	Creates a graph from a DGCNN File.
ByDGCNNPath(path[, key, tolerance])	Creates a graph from a DGCNN path.
ByDGCNNString(string[, key, tolerance])	Creates a graph from a DGCNN string.
ByIFCFile(file[, includeTypes, ...])	Create a Graph from an IFC file.
ByIFCPath(path[, includeTypes, ...])	Create a Graph from an IFC path.
ByMeshData(vertices, edges[, ...])	Creates a graph from the input mesh data
ByTopology(topology[, direct, ...])	Creates a graph.See https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).

continues on next page

Table 4 – continued from previous page

ByVerticesEdges(vertices, edges)	Creates a graph from the input list of vertices and edges.
ChromaticNumber(graph[, maxColors, silent])	Returns the chromatic number of the input graph.
ClosenessCentrality(graph[, vertices, key, ...])	Return the closeness centrality measure of the input list of vertices within the input graph.
Color(graph[, oldKey, key, maxColors, tolerance])	Colors the input vertices within the input graph.
Connect(graph, verticesA, verticesB[, tolerance])	Connects the two lists of input vertices.
ContainsEdge(graph, edge[, tolerance])	Returns True if the input graph contains the input edge.
ContainsVertex(graph, vertex[, tolerance])	Returns True if the input graph contains the input Vertex.
ContractEdge(graph, edge[, vertex, tolerance])	Contracts the input edge in the input graph into a single vertex.
Degree(graph[, vertices, key, edgeKey, ...])	Return the degree measure of the input list of vertices within the input graph.
DegreeSequence(graph)	Returns the degree sequence of the input graph.
Density(graph)	Returns the density of the input graph.
Depth(graph[, vertex, tolerance, silent])	Computes the maximum depth of the input graph rooted at the input vertex.
DepthMap(graph[, vertices, key, type, ...])	Return the depth map of the input list of vertices within the input graph.
Diameter(graph)	Returns the diameter of the input graph.
Dictionary(graph)	Returns the dictionary of the input graph.
Distance(graph, vertexA, vertexB[, type, ...])	Returns the shortest-path distance between the input vertices.
Edge(graph, vertexA, vertexB[, tolerance])	Returns the edge in the input graph that connects in the input vertices.
Edges(graph[, vertices, tolerance])	Returns the edges found in the input graph.
ExportToAdjacencyMatrixCSV(adjacencyMatrix, path)	Exports the input graph into a set of CSV files compatible with DGL.
ExportToBOT(graph, path[, format, ...])	Exports the input graph to an RDF graph serialized according to the BOT ontology.
ExportToCSV(graph, path, graphLabel[, ...])	Exports the input graph into a set of CSV files compatible with DGL.
ExportToGEXF(graph[, path, graphWidth, ...])	Exports the input graph to a Graph Exchange XML (GEXF) file format.
ExportToJSON(graph, path[, verticesKey, ...])	Exports the input graph to a JSON file.
GlobalClusteringCoefficient(graph)	Returns the global clustering coefficient of the input graph.
Guid(graph)	Returns the guid of the input graph
IncomingEdges(graph, vertex[, directed, ...])	Returns the incoming edges connected to a vertex.
IncomingVertices(graph, vertex[, directed, ...])	Returns the incoming vertices connected to a vertex.
IsBipartite(graph[, tolerance])	Returns True if the input graph is bipartite.
IsComplete(graph)	Returns True if the input graph is complete.
IsErdoesGallai(graph, sequence)	Returns True if the input sequence satisfies the Erdős–Gallai theorem.
IsTree(graph)	Returns True if the input graph has a hierarchical tree-like structure.
IsolatedVertices(graph)	Returns the list of isolated vertices in the input graph.
JSONData(graph[, verticesKey, edgesKey, ...])	Converts the input graph into JSON data.
JSONString(graph[, verticesKey, edgesKey, ...])	Converts the input graph into JSON data.

continues on next page

Table 4 – continued from previous page

LocalClusteringCoefficient(graph[, ...])	Returns the local clustering coefficient of the input list of vertices within the input graph.
LongestPath(graph, vertexA, vertexB[, ...])	Returns the longest path that connects the input vertices.
MaximumDelta(graph)	Returns the maximum delta of the input graph.
MaximumFlow(graph, source, sink[, ...])	Returns the maximum flow of the input graph.
MeshData(g[, tolerance])	Returns the mesh data of the input graph.
MetricDistance(graph, vertexA, vertexB[, ...])	Returns the shortest-path distance between the input vertices.
MinimumDelta(graph)	Returns the minimum delta of the input graph.
MinimumSpanningTree(graph[, edgeKey, tolerance])	Returns the minimum spanning tree of the input graph.
NavigationGraph(face[, sources, ...])	Creates a 2D navigation graph.
NearestVertex(graph, vertex)	Returns the vertex in the input graph that is the nearest to the input vertex.
NetworkXGraph(graph[, mantissa, tolerance])	converts the input graph into a NetworkX Graph.
Order(graph)	Returns the graph order of the input graph.
OutgoingEdges(graph, vertex[, directed, ...])	Returns the outgoing edges connected to a vertex.
OutgoingVertices(graph, vertex[, directed, ...])	Returns the list of outgoing vertices connected to a vertex.
PageRank(graph[, alpha, maxIterations, ...])	Calculates PageRank scores for nodes in a directed graph.
Path(graph, vertexA, vertexB[, tolerance])	Returns a path (wire) in the input graph that connects the input vertices.
PyvisGraph(graph, path[, overwrite, height, ...])	Displays a pyvis graph.
RemoveEdge(graph, edge[, tolerance])	Removes the input edge from the input graph.
RemoveVertex(graph, vertex[, tolerance])	Removes the input vertex from the input graph.
Reshape(graph[, shape, k, seed, iterations, ...])	Reshapes the input graph according to the desired input shape parameter.
SetDictionary(graph, dictionary)	Sets the input graph's dictionary to the input dictionary
ShortestPath(graph, vertexA, vertexB[, ...])	Returns the shortest path that connects the input vertices.
ShortestPaths(graph, vertexA, vertexB[, ...])	Returns the shortest path that connects the input vertices.
Show(graph[, sagitta, absolute, sides, ...])	Shows the graph using Plotly.
Size(graph)	Returns the graph size of the input graph.
TopologicalDistance(graph, vertexA, vertexB)	Returns the topological distance between the input vertices.
Topology(graph)	Returns the topology (cluster) of the input graph
Tree(graph[, vertex, tolerance])	Creates a tree graph version of the input graph rooted at the input vertex.
VertexDegree(graph, vertex[, edgeKey, tolerance])	Returns the degree of the input vertex.
Vertices(graph[, vertexKey, reverse])	Returns the list of vertices in the input graph.
VisibilityGraph(face[, viewpointsA, ...])	Creates a 2D visibility graph.

ByBOTGraph	
ByBOTPath	

static AddEdge(*graph*, *edge*, *transferVertexDictionaries: bool = False*, *transferEdgeDictionaries: bool = False*, *tolerance: float = 0.0001*, *silent: bool = False*)

Adds the input edge to the input Graph.

Parameters

graph
[topologic_core.Graph] The input graph.

edge
[topologic_core.Edge] The input edge.

transferVertexDictionaries
[bool, optional] If set to True, the dictionaries of the vertices are transferred to the graph.

transferEdgeDictionaries
[bool, optional] If set to True, the dictionaries of the edges are transferred to the graph.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Graph
The input graph with the input edge added to it.

`static AddVertex`(*graph*, *vertex*, *tolerance: float = 0.0001*, *silent: bool = False*)

Adds the input vertex to the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Graph
The input graph with the input vertex added to it.

`static AddVertices`(*graph*, *vertices*, *tolerance: float = 0.0001*, *silent: bool = False*)

Adds the input vertex to the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list] The input list of vertices.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Graph
The input graph with the input vertex added to it.

`AdjacencyDictionary`(*vertexLabelKey: str = 'label'*, *edgeKey: str = 'Length'*, *includeWeights: bool = False*, *reverse: bool = False*, *mantissa: int = 6*)

Returns the adjacency dictionary of the input Graph.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexLabelKey
[str , optional] The returned vertices are labelled according to the dictionary values stored under this key. If the vertexLabelKey does not exist, it will be created and the vertices are labelled numerically and stored in the vertex dictionary under this key. The default is "label".

edgeKey
[str , optional] If set, the edges' dictionaries will be searched for this key to set their weight. If the key is set to "length" (case insensitive), the length of the edge will be used as its weight. If set to None, a weight of 1 will be used. The default is "Length".

includeWeights
[bool , optional] If set to True, edge weights are included. Otherwise, they are not. The default is False.

reverse
[bool , optional] If set to True, the vertices are sorted in reverse order (only if vertexKey is set). Otherwise, they are not. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
The adjacency dictionary.

`static AdjacencyList`(*graph*, *vertexKey=None*, *reverse=True*, *tolerance=0.0001*)

Returns the adjacency list of the input Graph. See https://en.wikipedia.org/wiki/Adjacency_list.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexKey
[str , optional] If set, the returned list of vertices is sorted according to the dictionary values stored under this key. The default is None.

reverse
[bool , optional] If set to True, the vertices are sorted in reverse order (only if vertexKey is set). Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The adjacency list.

`static AdjacencyMatrix`(*graph*, *vertexKey=None*, *reverse=False*, *edgeKeyFwd=None*, *edgeKeyBwd=None*, *bidirKey=None*, *bidirectional=True*, *useEdgeIndex=False*, *useEdgeLength=False*, *tolerance=0.0001*)

Returns the adjacency matrix of the input Graph. See https://en.wikipedia.org/wiki/Adjacency_matrix.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexKey
[str , optional] If set, the returned list of vertices is sorted according to the dictionary values stored under this key. The default is None.

reverse
[bool , optional] If set to True, the vertices are sorted in reverse order (only if vertexKey is set). Otherwise, they are not. The default is False.

edgeKeyFwd
[str , optional] If set, the value at this key in the connecting edge from start vertex to end vertex (forward) will be used instead of the value 1. The default is None. useEdgeIndex and useEdgeLength override this setting.

edgeKeyBwd
[str , optional] If set, the value at this key in the connecting edge from end vertex to start vertex (backward) will be used instead of the value 1. The default is None. useEdgeIndex and useEdgeLength override this setting.

bidirKey
[bool , optional] If set to True or False, this key in the connecting edge will be used to determine is the edge is supposed to be bidirectional or not. If set to None, the input variable bidrectional will be used instead. The default is None

bidirectional
[bool , optional] If set to True, the edges in the graph that do not have a bidireKey in their dictionaries will be treated as being bidirectional. Otherwise, the start vertex and end vertex of the connecting edge will determine the direction. The default is True.

useEdgeIndex
[bool , False] If set to True, the adjacency matrix values will the index of the edge in Graph.Edges(graph). The default is False. Both useEdgeIndex, useEdgeLength should not be True at the same time. If they are, useEdgeLength will be used.

useEdgeLength
[bool , False] If set to True, the adjacency matrix values will the length of the edge in Graph.Edges(graph). The default is False. Both useEdgeIndex, useEdgeLength should not be True at the same time. If they are, useEdgeLength will be used.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The adjacency matrix.

`static AdjacentVertices`(*graph*, *vertex*, *silent: bool = False*)

Returns the list of vertices connected to the input vertex.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] the input vertex.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

list
The list of adjacent vertices.

`static AllPaths`(*graph*, *vertexA*, *vertexB*, *timeLimit=10*, *silent: bool = False*)

Returns all the paths that connect the input vertices within the allowed time limit in seconds.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

timeLimit
[int , optional] The time limit in second. The default is 10 seconds.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

list
The list of all paths (wires) found within the time limit.

`static AreIsomorphic`(*graphA*, *graphB*, *maxIterations=10*, *silent=False*)

Tests if the two input graphs are isomorphic according to the Weisfeiler Lehman graph isomorphism test. See https://en.wikipedia.org/wiki/Weisfeiler_Leman_graph_isomorphism_test

Parameters

graphA
[topologic_core.Graph] The first input graph.

graphB
[topologic_core.Graph] The second input graph.

maxIterations
[int , optional] This number limits the number of iterations to prevent the function from running indefinitely, particularly for very large or complex graphs.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

bool
True if the two input graphs are isomorphic. False otherwise

`static AverageClusteringCoefficient`(*graph*, *mantissa: int = 6*, *silent: bool = False*)

Returns the average clustering coefficient of the input graph. See https://en.wikipedia.org/wiki/Clustering_coefficient.

Parameters

graph
[topologic_core.Graph] The input graph.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The average clustering coefficient of the input graph.

`static BOTGraph`(*graph*, *bidirectional: bool = False*, *includeAttributes: bool = False*, *includeLabel: bool = False*, *includeGeometry: bool = False*, *siteLabel: str = 'Site_0001'*, *siteDictionary: dict = None*, *buildingLabel: str = 'Building_0001'*, *buildingDictionary: dict = None*, *storeyPrefix: str = 'Storey'*, *floorLevels: list = []*, *vertexLabelKey: str = 'label'*, *typeKey: str = 'type'*, *verticesKey: str = 'vertices'*, *edgesKey: str = 'edges'*, *edgeLabelKey: str = ''*, *sourceKey: str = 'source'*, *targetKey: str = 'target'*, *xKey: str = 'hasX'*, *yKey: str = 'hasY'*, *zKey: str = 'hasZ'*, *geometryKey: str = 'brep'*, *spaceType: str = 'space'*, *wallType: str = 'wall'*, *slabType: str = 'slab'*, *doorType: str = 'door'*, *windowType: str = 'window'*, *contentType: str = 'content'*, *namespace: str = 'http://github.com/wassimj/topologicpy/resources'*, *mantissa: int = 6*)

Creates an RDF graph according to the BOT ontology. See https://w3c-lbd-cg.github.io/bot/.

Parameters

graph
[topologic_core.Graph] The input graph.

bidirectional
[bool , optional] If set to True, reverse relationships are created wherever possible. Otherwise, they are not. The default is False.

includeAttributes
[bool , optional] If set to True, the attributes associated with vertices in the graph are written out. Otherwise, they are not. The default is False.

includeLabel
[bool , optional] If set to True, a label is attached to each node. Otherwise, it is not. The default is False.

includeGeometry
[bool , optional] If set to True, the geometry associated with vertices in the graph are written out. Otherwise, they are not. The default is False.

siteLabel
[str , optional] The desired site label. The default is "Site_0001".

siteDictionary
[dict , optional] The dictionary of site attributes to include in the output. The default is None.

buildingLabel
[str , optional] The desired building label. The default is "Building_0001".

buildingDictionary
[dict , optional] The dictionary of building attributes to include in the output. The default is None.

storeyPrefix
[str , optional] The desired prefixed to use for each building storey. The default is "Storey".

floorLevels
[list , optional] The list of floor levels. This should be a numeric list, sorted from lowest to highest. If not provided, floorLevels will be computed automatically based on the vertices' (zKey)) attribute. See below.

verticesKey
[str , optional] The desired key name to call vertices. The default is "vertices".

edgesKey
[str , optional] The desired key name to call edges. The default is "edges".

vertexLabelKey
[str , optional] If set to a valid string, the vertex label will be set to the value at this key. Otherwise it will be set to Vertex_XXXX where XXXX is a sequential unique number. Note: If vertex labels are not unique, they will be forced to be unique.

edgeLabelKey
[str , optional] If set to a valid string, the edge label will be set to the value at this key. Otherwise it will be set to Edge_XXXX where XXXX is a sequential unique number. Note: If edge labels are not unique, they will be forced to be unique.

sourceKey
[str , optional] The dictionary key used to store the source vertex. The default is "source".

targetKey
[str , optional] The dictionary key used to store the target vertex. The default is "target".

xKey
[str , optional] The desired key name to use for x-coordinates. The default is "hasX".

yKey
[str , optional] The desired key name to use for y-coordinates. The default is "hasY".

zKey
[str , optional] The desired key name to use for z-coordinates. The default is "hasZ".

geometryKey
[str , optional] The desired key name to use for geometry. The default is "brep".

typeKey
[str , optional] The dictionary key to use to look up the type of the node. The default is "type".

geometryKey
[str , optional] The dictionary key to use to look up the geometry of the node. The default is "brep".

spaceType
[str , optional] The dictionary string value to use to look up vertices of type "space". The default is "space".

wallType
[str , optional] The dictionary string value to use to look up vertices of type "wall". The default is "wall".

slabType
[str , optional] The dictionary string value to use to look up vertices of type "slab". The default is "slab".

doorType
[str , optional] The dictionary string value to use to look up vertices of type "door". The default is "door".

windowType
[str , optional] The dictionary string value to use to look up vertices of type "window". The default is "window".

contentType
[str , optional] The dictionary string value to use to look up vertices of type "content". The default is "contents".

namespace
[str , optional] The desired namespace to use in the BOT graph. The default is "http://github.com/wassimj/topologicpy/resources".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

rdflib.graph.Graph
The rdf graph using the BOT ontology.

`static BOTString`(*graph*, *format='turtle'*, *bidirectional: bool = False*, *includeAttributes: bool = False*, *includeLabel: bool = False*, *includeGeometry: bool = False*, *siteLabel: str = 'Site_0001'*, *siteDictionary: dict = None*, *buildingLabel: str = 'Building_0001'*, *buildingDictionary: dict = None*, *storeyPrefix: str = 'Storey'*, *floorLevels: list = []*, *vertexLabelKey: str = 'label'*, *typeKey: str = 'type'*, *verticesKey: str = 'vertices'*, *edgesKey: str = 'edges'*, *edgeLabelKey: str = ''*, *sourceKey: str = 'source'*, *targetKey: str = 'target'*, *xKey: str = 'hasX'*, *yKey: str = 'hasY'*, *zKey: str = 'hasZ'*, *geometryKey: str = 'brep'*, *spaceType: str = 'space'*, *wallType: str = 'wall'*, *slabType: str = 'slab'*, *doorType: str = 'door'*, *windowType: str = 'window'*, *contentType: str = 'content'*, *namespace: str = 'http://github.com/wassimj/topologicpy/resources'*, *mantissa: int = 6*)

Returns an RDF graph serialized string according to the BOT ontology. See https://w3c-lbd-cg.github.io/bot/.

Parameters

graph
[topologic_core.Graph] The input graph.

format
[str , optional] The desired output format, the options are listed below. Thde default is "turtle". turtle, ttl or turtle2 : Turtle, turtle2 is just turtle with more spacing & linebreaks xml or pretty-xml : RDF/XML, Was the default format, rdflib < 6.0.0 json-ld : JSON-LD , There are further options for compact syntax and other JSON-LD variants ntriples, nt or nt11 : N-Triples , nt11 is exactly like nt, only utf8 encoded n3 : Notation-3 , N3 is a superset of Turtle that also caters for rules and a few other things trig : Trig , Turtle-like format for RDF triples + context (RDF quads) and thus multiple graphs trix : Trix , RDF/XML-like format for RDF quads nquads : N-Quads , N-Triples-like format for RDF quads

bidirectional
[bool , optional] If set to True, reverse relationships are created wherever possible. Otherwise, they are not. The default is False.

includeAttributes
[bool , optional] If set to True, the attributes associated with vertices in the graph are written out. Otherwise, they are not. The default is False.

includeLabel
[bool , optional] If set to True, a label is attached to each node. Otherwise, it is not. The default is False.

includeGeometry
[bool , optional] If set to True, the geometry associated with vertices in the graph are written out. Otherwise, they are not. The default is False.

siteLabel
[str , optional] The desired site label. The default is "Site_0001".

siteDictionary
[dict , optional] The dictionary of site attributes to include in the output. The default is None.

buildingLabel
[str , optional] The desired building label. The default is "Building_0001".

buildingDictionary
[dict , optional] The dictionary of building attributes to include in the output. The default is None.

storeyPrefix
[str , optional] The desired prefixed to use for each building storey. The default is "Storey".

floorLevels
[list , optional] The list of floor levels. This should be a numeric list, sorted from lowest to highest. If not provided, floorLevels will be computed automatically based on the vertices' (zKey)) attribute. See below.

verticesKey
[str , optional] The desired key name to call vertices. The default is "vertices".

edgesKey
[str , optional] The desired key name to call edges. The default is "edges".

vertexLabelKey
[str , optional] If set to a valid string, the vertex label will be set to the value at this key. Otherwise it will be set to Vertex_XXXX where XXXX is a sequential unique number. Note: If vertex labels are not unique, they will be forced to be unique.

edgeLabelKey
[str , optional] If set to a valid string, the edge label will be set to the value at this key.

Otherwise it will be set to Edge_XXXX where XXXX is a sequential unique number. Note: If edge labels are not unique, they will be forced to be unique.

sourceKey
[str , optional] The dictionary key used to store the source vertex. The default is "source".

targetKey
[str , optional] The dictionary key used to store the target vertex. The default is "target".

xKey
[str , optional] The desired key name to use for x-coordinates. The default is "hasX".

yKey
[str , optional] The desired key name to use for y-coordinates. The default is "hasY".

zKey
[str , optional] The desired key name to use for z-coordinates. The default is "hasZ".

geometryKey
[str , optional] The desired key name to use for geometry. The default is "brep".

typeKey
[str , optional] The dictionary key to use to look up the type of the node. The default is "type".

spaceType
[str , optional] The dictionary string value to use to look up vertices of type "space". The default is "space".

wallType
[str , optional] The dictionary string value to use to look up vertices of type "wall". The default is "wall".

slabType
[str , optional] The dictionary string value to use to look up vertices of type "slab". The default is "slab".

doorType
[str , optional] The dictionary string value to use to look up vertices of type "door". The default is "door".

windowType
[str , optional] The dictionary string value to use to look up vertices of type "window". The default is "window".

contentType
[str , optional] The dictionary string value to use to look up vertices of type "content". The default is "contents".

namespace
[str , optional] The desired namespace to use in the BOT graph. The default is "http://github.com/wassimj/topologicpy/resources".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

str
The rdf graph serialized string using the BOT ontology.

static BetweenessCentrality(*graph*, *vertices=None*, *sources=None*, *destinations=None*, *key: str = 'betweeness_centrality'*, *mantissa: int = 6*, *tolerance: float = 0.001*)

Returns the betweeness centrality measure of the input list of vertices within the input graph. The order of the returned list is the same as the order of the input list of vertices. If no vertices are specified, the betweeness centrality of all the vertices in the input graph is computed. See https://en.wikipedia.org/wiki/Betweenness_centrality.

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list , optional] The input list of vertices. The default is None which means all vertices in the input graph are considered.

sources
[list , optional] The input list of source vertices. The default is None which means all vertices in the input graph are considered.

destinations
[list , optional] The input list of destination vertices. The default is None which means all vertices in the input graph are considered.

key
[str , optional] The dictionary key under which to save the betweeness centrality score. The default is "betweneess_centrality".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The betweeness centrality of the input list of vertices within the input graph. The values are in the range 0 to 1.

static ByAdjacencyMatrix(*adjacencyMatrix*, *dictionaries=None*, *xMin=-0.5*, *yMin=-0.5*, *zMin=-0.5*, *xMax=0.5*, *yMax=0.5*, *zMax=0.5*, *silent=False*)

Returns graphs according to the input folder path. This method assumes the CSV files follow DGL's schema.

Parameters

adjacencyMatrix
[list] The adjacency matrix expressed as a nested list of 0s and 1s.

dictionaries
[list , optional] A list of dictionaries to assign to the vertices of the graph. This list should be in the same order and of the same length as the rows in the adjacency matrix.

xMin
[float , optional] The desired minimum value to assign for a vertex's X coordinate. The default is -0.5.

yMin
[float , optional] The desired minimum value to assign for a vertex's Y coordinate. The default is -0.5.

zMin
[float , optional] The desired minimum value to assign for a vertex's Z coordinate. The default is -0.5.

xMax
[float , optional] The desired maximum value to assign for a vertex's X coordinate. The default is 0.5.

yMax
[float , optional] The desired maximum value to assign for a vertex's Y coordinate. The default is 0.5.

zMax
[float , optional] The desired maximum value to assign for a vertex's Z coordinate. The default is 0.5.

silent
[bool , optional] If set to True, no warnings or error messages are displayed. The default is False.

Returns

topologic_core.Graph
The created graph.

`static ByAdjacencyMatrixCSVPath`(*path: str*, *dictionaries: list = None*, *silent: bool = False*)

Returns graphs according to the input path. This method assumes the CSV files follow an adjacency matrix schema.

Parameters

path
[str] The file path to the adjacency matrix CSV file.

dictionaries
[list , optional] A list of dictionaries to assign to the vertices of the graph. This list should be in the same order and of the same length as the rows in the adjacency matrix.

silent
[bool , optional] If set to True, no warnings or error messages are displayed. The default is False.

Returns

topologic_core.Graph
The created graph.

`static ByBOTGraph`(*botGraph*, *includeContext=False*, *xMin=-0.5*, *xMax=0.5*, *yMin=-0.5*, *yMax=0.5*, *zMin=-0.5*, *zMax=0.5*, *tolerance=0.0001*)

`static ByBOTPath`(*path*, *includeContext=False*, *xMin=-0.5*, *xMax=0.5*, *yMin=-0.5*, *yMax=0.5*, *zMin=-0.5*, *zMax=0.5*, *tolerance=0.0001*)

static ByCSVPath(*path*, *graphIDHeader='graph_id'*, *graphLabelHeader='label'*, *graphFeaturesHeader='feat'*, *graphFeaturesKeys=[]*, *edgeSRCHeader='src_id'*, *edgeDSTHeader='dst_id'*, *edgeLabelHeader='label'*, *edgeTrainMaskHeader='train_mask'*, *edgeValidateMaskHeader='val_mask'*, *edgeTestMaskHeader='test_mask'*, *edgeFeaturesHeader='feat'*, *edgeFeaturesKeys=[]*, *nodeIDHeader='node_id'*, *nodeLabelHeader='label'*, *nodeTrainMaskHeader='train_mask'*, *nodeValidateMaskHeader='val_mask'*, *nodeTestMaskHeader='test_mask'*, *nodeFeaturesHeader='feat'*, *nodeXHeader='X'*, *nodeYHeader='Y'*, *nodeZHeader='Z'*, *nodeFeaturesKeys=[]*, *tolerance=0.0001*, *silent=False*)

Returns graphs according to the input folder path. This method assumes the CSV files follow DGL's schema.

Parameters

path
[str] The path to the folder containing the .yaml and .csv files for graphs, edges, and nodes.

graphIDHeader
[str , optional] The column header string used to specify the graph id. The default is "graph_id".

graphLabelHeader
[str , optional] The column header string used to specify the graph label. The default is "label".

graphFeaturesHeader
[str , optional] The column header string used to specify the graph features. The default is "feat".

edgeSRCHeader
[str , optional] The column header string used to specify the source vertex id of edges. The default is "src_id".

edgeDSTHeader
[str , optional] The column header string used to specify the destination vertex id of edges. The default is "dst_id".

edgeLabelHeader
[str , optional] The column header string used to specify the label of edges. The default is "label".

edgeTrainMaskHeader
[str , optional] The column header string used to specify the train mask of edges. The default is "train_mask".

edgeValidateMaskHeader
[str , optional] The column header string used to specify the validate mask of edges. The default is "val_mask".

edgeTestMaskHeader
[str , optional] The column header string used to specify the test mask of edges. The default is "test_mask".

edgeFeaturesHeader
[str , optional] The column header string used to specify the features of edges. The default is "feat".

edgeFeaturesKeys
[list , optional] The list of dictionary keys to use to index the edge features. The length of this list must match the length of edge features. The default is [].

nodeIDHeader
[str , optional] The column header string used to specify the id of nodes. The default is "node_id".

nodeLabelHeader
[str , optional] The column header string used to specify the label of nodes. The default is "label".

nodeTrainMaskHeader
[str , optional] The column header string used to specify the train mask of nodes. The default is "train_mask".

nodeValidateMaskHeader
[str , optional] The column header string used to specify the validate mask of nodes. The default is "val_mask".

nodeTestMaskHeader
[str , optional] The column header string used to specify the test mask of nodes. The default is "test_mask".

nodeFeaturesHeader
[str , optional] The column header string used to specify the features of nodes. The default is "feat".

nodeXHeader
[str , optional] The column header string used to specify the X coordinate of nodes. The default is "X".

nodeYHeader
[str , optional] The column header string used to specify the Y coordinate of nodes. The default is "Y".

nodeZHeader
[str , optional] The column header string used to specify the Z coordinate of nodes. The default is "Z".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

dict
The dictionary of DGL graphs and labels found in the input CSV files. The keys in the dictionary are "graphs", "labels", "features"

`static ByDGCNNFile`(*file*, *key: str = 'label'*, *tolerance: float = 0.0001*)

Creates a graph from a DGCNN File.

Parameters

file
[file object] The input file.

key
[str , optional] The desired key for storing the node label. The default is "label".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dict
A dictionary with the graphs and labels. The keys are 'graphs' and 'labels'.

`static ByDGCNNPath`(*path*, *key: str = 'label'*, *tolerance: float = 0.0001*)

Creates a graph from a DGCNN path.

Parameters

path
[str] The input file path.

key
[str , optional] The desired key for storing the node label. The default is "label".

tolerance
[str , optional] The desired tolerance. The default is 0.0001.

Returns

dict
A dictionary with the graphs and labels. The keys are 'graphs' and 'labels'.

`static ByDGCNNString`(*string*, *key: str = 'label'*, *tolerance: float = 0.0001*)

Creates a graph from a DGCNN string.

Parameters

string
[str] The input string.

key
[str , optional] The desired key for storing the node label. The default is "label".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dict
A dictionary with the graphs and labels. The keys are 'graphs' and 'labels'.

`static ByIFCFile`(*file*, *includeTypes: list = []*, *excludeTypes: list = []*, *includeRels: list = []*, *excludeRels: list = []*, *xMin: float = -0.5*, *yMin: float = -0.5*, *zMin: float = -0.5*, *xMax: float = 0.5*, *yMax: float = 0.5*, *zMax: float = 0.5*, *tolerance: float = 0.0001*)

Create a Graph from an IFC file. This code is partially based on code from Bruno Postle.

Parameters

file
[file] The input IFC file

includeTypes
[list , optional] A list of IFC object types to include in the graph. The default is [] which means all object types are included.

excludeTypes
[list , optional] A list of IFC object types to exclude from the graph. The default is [] which mean no object type is excluded.

includeRels
[list , optional] A list of IFC relationship types to include in the graph. The default is [] which means all relationship types are included.

excludeRels
[list , optional] A list of IFC relationship types to exclude from the graph. The default is [] which mean no relationship type is excluded.

xMin
[float, optional] The desired minimum value to assign for a vertex's X coordinate. The default is -0.5.

yMin
[float, optional] The desired minimum value to assign for a vertex's Y coordinate. The default is -0.5.

zMin
[float, optional] The desired minimum value to assign for a vertex's Z coordinate. The default is -0.5.

xMax
[float, optional] The desired maximum value to assign for a vertex's X coordinate. The default is 0.5.

yMax
[float, optional] The desired maximum value to assign for a vertex's Y coordinate. The default is 0.5.

zMax
[float, optional] The desired maximum value to assign for a vertex's Z coordinate. The default is 0.5.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The created graph.

`static ByIFCPath`(*path*, *includeTypes=[]*, *excludeTypes=[]*, *includeRels=[]*, *excludeRels=[]*, *xMin=-0.5*, *yMin=-0.5*, *zMin=-0.5*, *xMax=0.5*, *yMax=0.5*, *zMax=0.5*)

Create a Graph from an IFC path. This code is partially based on code from Bruno Postle.

Parameters

path
[str] The input IFC file path.

includeTypes
[list , optional] A list of IFC object types to include in the graph. The default is [] which means all object types are included.

excludeTypes
[list , optional] A list of IFC object types to exclude from the graph. The default is [] which mean no object type is excluded.

includeRels
[list , optional] A list of IFC relationship types to include in the graph. The default is [] which means all relationship types are included.

excludeRels
[list , optional] A list of IFC relationship types to exclude from the graph. The default is [] which mean no relationship type is excluded.

xMin
[float, optional] The desired minimum value to assign for a vertex's X coordinate. The default is -0.5.

yMin
[float, optional] The desired minimum value to assign for a vertex's Y coordinate. The default is -0.5.

zMin
[float, optional] The desired minimum value to assign for a vertex's Z coordinate. The default is -0.5.

xMax
[float, optional] The desired maximum value to assign for a vertex's X coordinate. The default is 0.5.

yMax
[float, optional] The desired maximum value to assign for a vertex's Y coordinate. The default is 0.5.

zMax
[float, optional] The desired maximum value to assign for a vertex's Z coordinate. The default is 0.5.

Returns

topologic_core.Graph
The created graph.

`static ByMeshData`(*vertices*, *edges*, *vertexDictionaries=None*, *edgeDictionaries=None*, *tolerance=0.0001*)

Creates a graph from the input mesh data

Parameters

vertices
[list] The list of [x, y, z] coordinates of the vertices/

edges
[list] the list of [i, j] indices into the vertices list to signify and edge that connects vertices[i] to vertices[j].

vertexDictionaries
[list , optional] The python dictionaries of the vertices (in the same order as the list of vertices).

edgeDictionaries
[list , optional] The python dictionaries of the edges (in the same order as the list of edges).

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The created graph

`static ByTopology`(*topology*, *direct: bool = True*, *directApertures: bool = False*, *viaSharedTopologies: bool = False*, *viaSharedApertures: bool = False*, *toExteriorTopologies: bool = False*, *toExteriorApertures: bool = False*, *toContents: bool = False*, *toOutposts: bool = False*, *idKey: str = 'TOPOLOGIC_ID'*, *outpostsKey: str = 'outposts'*, *vertexCategoryKey: str = 'category'*, *edgeCategoryKey: str = 'category'*, *useInternalVertex: bool = True*, *storeBREP: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a graph.See https://en.wikipedia.org/wiki/Graph_(discrete_mathematics).

Parameters

topology
[topologic_core.Topology] The input topology.

direct
[bool , optional] If set to True, connect the subtopologies directly with a single edge. The default is True.

directApertures
[bool , optional] If set to True, connect the subtopologies directly with a single edge if they share one or more apertures. The default is False.

viaSharedTopologies
[bool , optional] If set to True, connect the subtopologies via their shared topologies. The default is False.

viaSharedApertures
[bool , optional] If set to True, connect the subtopologies via their shared apertures. The default is False.

toExteriorTopologies
[bool , optional] If set to True, connect the subtopologies to their exterior topologies. The default is False.

toExteriorApertures
[bool , optional] If set to True, connect the subtopologies to their exterior apertures. The default is False.

toContents
[bool , optional] If set to True, connect the subtopologies to their contents. The default is False.

toOutposts
[bool , optional] If set to True, connect the topology to the list specified in its outposts. The default is False.

idKey
[str , optional] The key to use to find outpost by ID. It is case insensitive. The default is "TOPOLOGIC_ID".

outpostsKey
[str , optional] The key to use to find the list of outposts. It is case insensitive. The default is "outposts".

vertexCategoryKey
[str , optional] The key under which to store the node type. Node categories are: 0 : main topology 1 : shared topology 2 : shared aperture 3 : exterior topology 4 : exterior aperture 5 : content 6 : outpost The default is "category".

edgeCategoryKey
[str , optional] The key under which to store the node type. Edge categories are: 0 : direct 1 : via shared topology 2 : via shared aperture 3 : to exterior topology 4 : to exterior aperture 5 : to content 6 : to outpost The default is "category".

useInternalVertex
[bool , optional] If set to True, use an internal vertex to represent the subtopology. Otherwise, use its centroid. The default is False.

storeBREP
[bool , optional] If set to True, store the BRep of the subtopology in its representative vertex. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The created graph.

`static ByVerticesEdges`(*vertices*, *edges*)

Creates a graph from the input list of vertices and edges.

Parameters

vertices
[list] The input list of vertices.

edges
[list] The input list of edges.

Returns

topologic_core.Graph
The created graph.

`static ChromaticNumber`(*graph*, *maxColors: int = 3*, *silent: bool = False*)

Returns the chromatic number of the input graph. See https://en.wikipedia.org/wiki/Graph_coloring.

Parameters

graph
[topologic_core.Graph] The input graph.

maxColors
[int , optional] The desired maximum number of colors to test against. The default is 3.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

int
The chromatic number of the input graph.

`static ClosenessCentrality`(*graph*, *vertices=None*, *key: str = 'closeness_centrality'*, *mantissa: int = 6*, *tolerance=0.0001*)

Return the closeness centrality measure of the input list of vertices within the input graph. The order of the returned list is the same as the order of the input list of vertices. If no vertices are specified, the closeness centrality of all the vertices in the input graph is computed. See https://en.wikipedia.org/wiki/Closeness_centrality.

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list , optional] The input list of vertices. The default is None.

key
[str , optional] The dictionary key under which to save the closeness centrality score. The default is "closeness_centrality".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The closeness centrality of the input list of vertices within the input graph. The values are in the range 0 to 1.

`static Color`(*graph*, *oldKey: str = 'color'*, *key: str = 'color'*, *maxColors: int = None*, *tolerance: float = 0.0001*)

Colors the input vertices within the input graph. The saved value is an integer rather than an actual color. See Color.ByValueInRange to convert to an actual color. Any vertices that have been pre-colored will not be affected. See https://en.wikipedia.org/wiki/Graph_coloring.

Parameters

graph
[topologic_core.Graph] The input graph.

oldKey
[str , optional] The existing dictionary key to use to read any pre-existing color information. The default is "color".

key
[str , optional] The new dictionary key to use to write out new color information. The default is "color".

maxColors
[int , optional] The desired maximum number of colors to use. If set to None, the chromatic number of the graph is used. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The input graph, but with its vertices colored.

`static Connect`(*graph*, *verticesA*, *verticesB*, *tolerance=0.0001*)

Connects the two lists of input vertices.

Parameters

graph
[topologic_core.Graph] The input graph.

verticesA
[list] The first list of input vertices.

verticesB
[topologic_core.Vertex] The second list of input vertices.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The input graph with the connected input vertices.

`static ContainsEdge`(*graph*, *edge*, *tolerance=0.0001*)

Returns True if the input graph contains the input edge. Returns False otherwise.

Parameters

graph
[topologic_core.Graph] The input graph.

edge
[topologic_core.Edge] The input edge.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the input graph contains the input edge. False otherwise.

`static ContainsVertex`(*graph*, *vertex*, *tolerance=0.0001*)

Returns True if the input graph contains the input Vertex. Returns False otherwise.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input Vertex.

tolerance
[float , optional] Ther desired tolerance. The default is 0.0001.

Returns

bool
True if the input graph contains the input vertex. False otherwise.

`static ContractEdge`(*graph*, *edge*, *vertex=None*, *tolerance=0.0001*)

Contracts the input edge in the input graph into a single vertex. Please note that the dictionary of the edge is transferred to the vertex that replaces it. See https://en.wikipedia.org/wiki/Edge_contraction

Parameters

graph
[topologic_core.Graph] The input graph.

edge
[topologic_core.Edge] The input graph edge that needs to be contracted.

vertex
[topollogic.Vertex , optional] The vertex to replace the contracted edge. If set to None, the centroid of the edge is chosen. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The input graph, but with input edge contracted into a single vertex.

`static Degree`(*graph*, *vertices=None*, *key: str = 'degree'*, *edgeKey: str = None*, *mantissa: int = 6*, *tolerance=0.0001*)

Return the degree measure of the input list of vertices within the input graph. The order of the returned list is the same as the order of the input list of vertices. If no vertices are specified, the closeness centrality of all the vertices in the input graph is computed. See https://en.wikipedia.org/wiki/Degree_(graph_theory).

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list , optional] The input list of vertices. The default is None.

key
[str , optional] The dictionary key under which to save the closeness centrality score. The default is "degree".

edgeKey
[str , optional] If specified, the value in the connected edges' dictionary specified by the edgeKey string will be aggregated to calculate the vertex degree. If a numeric value cannot be retrieved from an edge, a value of 1 is used instead. This is used in weighted graphs.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The degree of the input list of vertices within the input graph.

`static DegreeSequence`(*graph*)

Returns the degree sequence of the input graph. See https://mathworld.wolfram.com/DegreeSequence.html.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

list
The degree sequence of the input graph.

`static Density`(*graph*)

Returns the density of the input graph. See https://en.wikipedia.org/wiki/Dense_graph.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

float
The density of the input graph.

static Depth(*graph*, *vertex=None*, *tolerance: float = 0.0001*, *silent: bool = False*)

Computes the maximum depth of the input graph rooted at the input vertex.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex , optional] The input root vertex. If not set, the first vertex in the graph is set as the root vertex. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

int
The calculated maximum depth of the input graph rooted at the input vertex.

static DepthMap(*graph*, *vertices=None*, *key: str = 'depth'*, *type: str = 'topological'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Return the depth map of the input list of vertices within the input graph. The returned list contains the total of the topological distances of each vertex to every other vertex in the input graph. The order of the depth map list is the same as the order of the input list of vertices. If no vertices are specified, the depth map of all the vertices in the input graph is computed.

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list , optional] The input list of vertices. The default is None.

key
[str , optional] The dictionary key under which to save the depth score. The default is "depth".

type
[str , optional] The type of depth distance to calculate. The options are "topological" or "metric". The default is "topological". See https://www.spacesyntax.online/overview-2/analysis-of-spatial-relations/.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The depth map of the input list of vertices within the input graph.

static Diameter(*graph*)

Returns the diameter of the input graph. See https://mathworld.wolfram.com/GraphDiameter.html.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

int
The diameter of the input graph.

`static Dictionary`(*graph*)

Returns the dictionary of the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

topologic_core.Dictionary
The dictionary of the input graph.

`static Distance`(*graph*, *vertexA*, *vertexB*, *type: str = 'topological'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the shortest-path distance between the input vertices. See https://en.wikipedia.org/wiki/Distance_(graph_theory).

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

type
[str , optional] The type of depth distance to calculate. The options are "topological" or "metric". The default is "topological". See https://www.spacesyntax.online/overview-2/analysis-of-spatial-relations/.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

float
The shortest-path metric distance between the input vertices.

`static Edge`(*graph*, *vertexA*, *vertexB*, *tolerance=0.0001*)

Returns the edge in the input graph that connects in the input vertices.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input Vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Edge
The edge in the input graph that connects the input vertices.

`static Edges`(*graph*, *vertices=None*, *tolerance=0.0001*)

Returns the edges found in the input graph. If the input list of vertices is specified, this method returns the edges connected to this list of vertices. Otherwise, it returns all graph edges.

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list , optional] An optional list of vertices to restrict the returned list of edges only to those connected to this list.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of edges in the graph.

`static ExportToAdjacencyMatrixCSV`(*adjacencyMatrix*, *path*)

Exports the input graph into a set of CSV files compatible with DGL.

Parameters

adjacencyMatrix: list
The input adjacency matrix.

path
[str] The desired path to the output folder where the graphs, edges, and nodes CSV files will be saved.

Returns

bool
True if the graph has been successfully exported. False otherwise.

`static ExportToBOT`(*graph*, *path: str*, *format: str = 'turtle'*, *overwrite: bool = False*, *bidirectional: bool = False*, *includeAttributes: bool = False*, *includeLabel: bool = False*, *includeGeometry: bool = False*, *siteLabel: str = 'Site_0001'*, *siteDictionary: dict = None*, *buildingLabel: str = 'Building_0001'*, *buildingDictionary: dict = None*, *storeyPrefix: str = 'Storey'*, *floorLevels: list = []*, *vertexLabelKey: str = 'label'*, *typeKey: str = 'type'*, *verticesKey: str = 'vertices'*, *edgesKey: str = 'edges'*, *edgeLabelKey: str = ''*, *sourceKey: str = 'source'*, *targetKey: str = 'target'*, *xKey: str = 'hasX'*, *yKey: str = 'hasY'*, *zKey: str = 'hasZ'*, *geometryKey: str = 'brep'*, *spaceType: str = 'space'*, *wallType: str = 'wall'*, *slabType: str = 'slab'*, *doorType: str = 'door'*, *windowType: str = 'window'*, *contentType: str = 'content'*, *namespace: str = 'http://github.com/wassimj/topologicpy/resources'*, *mantissa: int = 6*)

Exports the input graph to an RDF graph serialized according to the BOT ontology. See https://w3c-lbd-cg.github.io/bot/.

Parameters

graph
[topologic_core.Graph] The input graph.

path
[str] The desired path to where the RDF/BOT file will be saved.

format
[str , optional] The desired output format, the options are listed below. Thde default is "turtle". turtle, ttl or turtle2 : Turtle, turtle2 is just turtle with more spacing & linebreaks xml or pretty-xml : RDF/XML, Was the default format, rdflib < 6.0.0 json-ld : JSON-LD , There are further options for compact syntax and other JSON-LD variants ntriples, nt or nt11 : N-Triples , nt11 is exactly like nt, only utf8 encoded n3 : Notation-3 , N3 is a superset of Turtle that also caters for rules and a few other things trig : Trig , Turtle-like format for RDF triples + context (RDF quads) and thus multiple graphs trix : Trix , RDF/XML-like format for RDF quads nquads : N-Quads , N-Triples-like format for RDF quads

overwrite
[bool , optional] If set to True, any existing file is overwritten. Otherwise, it is not. The default is False.

bidirectional
[bool , optional] If set to True, reverse relationships are created wherever possible. Otherwise, they are not. The default is False.

includeAttributes
[bool , optional] If set to True, the attributes associated with vertices in the graph are written out. Otherwise, they are not. The default is False.

includeLabel
[bool , optional] If set to True, a label is attached to each node. Otherwise, it is not. The default is False.

includeGeometry
[bool , optional] If set to True, the geometry associated with vertices in the graph are written out. Otherwise, they are not. The default is False.

siteLabel
[str , optional] The desired site label. The default is "Site_0001".

siteDictionary
[dict , optional] The dictionary of site attributes to include in the output. The default is None.

buildingLabel
[str , optional] The desired building label. The default is "Building_0001".

buildingDictionary
[dict , optional] The dictionary of building attributes to include in the output. The default is None.

storeyPrefix
[str , optional] The desired prefixed to use for each building storey. The default is "Storey".

floorLevels
[list , optional] The list of floor levels. This should be a numeric list, sorted from lowest to highest. If not provided, floorLevels will be computed automatically based on the vertices' (zKey)) attribute. See below.

verticesKey
[str , optional] The desired key name to call vertices. The default is “vertices”.

edgesKey
[str , optional] The desired key name to call edges. The default is “edges”.

vertexLabelKey
[str , optional] If set to a valid string, the vertex label will be set to the value at this key. Otherwise it will be set to Vertex_XXXX where XXXX is a sequential unique number. Note: If vertex labels are not unique, they will be forced to be unique.

edgeLabelKey
[str , optional] If set to a valid string, the edge label will be set to the value at this key. Otherwise it will be set to Edge_XXXX where XXXX is a sequential unique number. Note: If edge labels are not unique, they will be forced to be unique.

sourceKey
[str , optional] The dictionary key used to store the source vertex. The default is “source”.

targetKey
[str , optional] The dictionary key used to store the target vertex. The default is “target”.

xKey
[str , optional] The desired key name to use for x-coordinates. The default is “hasX”.

yKey
[str , optional] The desired key name to use for y-coordinates. The default is “hasY”.

zKey
[str , optional] The desired key name to use for z-coordinates. The default is “hasZ”.

geometryKey
[str , optional] The desired key name to use for geometry. The default is “brep”.

typeKey
[str , optional] The dictionary key to use to look up the type of the node. The default is “type”.

geometryKey
[str , optional] The dictionary key to use to look up the geometry of the node. The default is “brep”.

spaceType
[str , optional] The dictionary string value to use to look up vertices of type “space”. The default is “space”.

wallType
[str , optional] The dictionary string value to use to look up vertices of type “wall”. The default is “wall”.

slabType
[str , optional] The dictionary string value to use to look up vertices of type “slab”. The default is “slab”.

doorType
[str , optional] The dictionary string value to use to look up vertices of type “door”. The default is “door”.

windowType
[str , optional] The dictionary string value to use to look up vertices of type “window”. The default is “window”.

contentType
[str , optional] The dictionary string value to use to look up vertices of type "content". The default is "contents".

namespace
[str , optional] The desired namespace to use in the BOT graph. The default is "http://github.com/wassimj/topologicpy/resources".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

str
The rdf graph serialized string using the BOT ontology.

`static ExportToCSV`(*graph*, *path*, *graphLabel*, *graphFeatures=''*, *graphIDHeader='graph_id'*, *graphLabelHeader='label'*, *graphFeaturesHeader='feat'*, *edgeLabelKey='label'*, *defaultEdgeLabel=0*, *edgeFeaturesKeys=[]*, *edgeSRCHeader='src_id'*, *edgeDSTHeader='dst_id'*, *edgeLabelHeader='label'*, *edgeFeaturesHeader='feat'*, *edgeTrainMaskHeader='train_mask'*, *edgeValidateMaskHeader='val_mask'*, *edgeTestMaskHeader='test_mask'*, *edgeMaskKey='mask'*, *edgeTrainRatio=0.8*, *edgeValidateRatio=0.1*, *edgeTestRatio=0.1*, *bidirectional=True*, *nodeLabelKey='label'*, *defaultNodeLabel=0*, *nodeFeaturesKeys=[]*, *nodeIDHeader='node_id'*, *nodeLabelHeader='label'*, *nodeFeaturesHeader='feat'*, *nodeTrainMaskHeader='train_mask'*, *nodeValidateMaskHeader='val_mask'*, *nodeTestMaskHeader='test_mask'*, *nodeMaskKey='mask'*, *nodeTrainRatio=0.8*, *nodeValidateRatio=0.1*, *nodeTestRatio=0.1*, *mantissa=6*, *tolerance=0.0001*, *overwrite=False*)

Exports the input graph into a set of CSV files compatible with DGL.

Parameters

graph
[topologic_core.Graph] The input graph

path
[str] The desired path to the output folder where the graphs, edges, and nodes CSV files will be saved.

graphLabel
[float or int] The input graph label. This can be an int (categorical) or a float (continous)

graphFeatures
[str , optional] The input graph features. This is a single string of numeric features separated by commas. Example: "3.456, 2.011, 56.4". The defauly is "".

graphIDHeader
[str , optional] The desired graph ID column header. The default is "graph_id".

graphLabelHeader
[str , optional] The desired graph label column header. The default is "label".

graphFeaturesHeader
[str , optional] The desired graph features column header. The default is "feat".

edgeLabelKey
[str , optional] The edge label dictionary key saved in each graph edge. The default is "label".

defaultEdgeLabel
[int , optional] The default edge label to use if no edge label is found. The default is 0.

edgeLabelHeader
[str , optional] The desired edge label column header. The default is "label".

edgeSRCHeader
[str , optional] The desired edge source column header. The default is "src_id".

edgeDSTHeader
[str , optional] The desired edge destination column header. The default is "dst_id".

edgeFeaturesHeader
[str , optional] The desired edge features column header. The default is "feat".

edgeFeaturesKeys
[list , optional] The list of feature dictionary keys saved in the dicitonaries of edges. The default is [].

edgeTrainMaskHeader
[str , optional] The desired edge train mask column header. The default is "train_mask".

edgeValidateMaskHeader
[str , optional] The desired edge validate mask column header. The default is "val_mask".

edgeTestMaskHeader
[str , optional] The desired edge test mask column header. The default is "test_mask".

edgeMaskKey
[str , optional] The dictionary key where the edge train, validate, test category is to be found. The value should be 0 for train 1 for validate, and 2 for test. If no key is found, the ratio of train/validate/test will be used. The default is "mask".

edgeTrainRatio
[float , optional] The desired ratio of the edge data to use for training. The number must be between 0 and 1. The default is 0.8 which means 80% of the data will be used for training. This value is ignored if an edgeMaskKey is foud.

edgeValidateRatio
[float , optional] The desired ratio of the edge data to use for validation. The number must be between 0 and 1. The default is 0.1 which means 10% of the data will be used for validation. This value is ignored if an edgeMaskKey is foud.

edgeTestRatio
[float , optional] The desired ratio of the edge data to use for testing. The number must be between 0 and 1. The default is 0.1 which means 10% of the data will be used for testing. This value is ignored if an edgeMaskKey is foud.

bidirectional
[bool , optional] If set to True, a reversed edge will also be saved for each edge in the graph. Otherwise, it will not. The default is True.

nodeFeaturesKeys
[list , optional] The list of features keys saved in the dicitonaries of nodes. The default is [].

nodeLabelKey
[str , optional] The node label dictionary key saved in each graph vertex. The default is "label".

defaultNodeLabel
[int , optional] The default node label to use if no node label is found. The default is 0.

nodeIDHeader
[str , optional] The desired node ID column header. The default is "node_id".

nodeLabelHeader
[str , optional] The desired node label column header. The default is "label".

nodeFeaturesHeader
[str , optional] The desired node features column header. The default is "feat".

nodeTrainMaskHeader
[str , optional] The desired node train mask column header. The default is "train_mask".

nodeValidateMaskHeader
[str , optional] The desired node validate mask column header. The default is "val_mask".

nodeTestMaskHeader
[str , optional] The desired node test mask column header. The default is "test_mask".

nodeMaskKey
[str , optional] The dictionary key where the node train, validate, test category is to be found. The value should be 0 for train 1 for validate, and 2 for test. If no key is found, the ratio of train/validate/test will be used. The default is "mask".

nodeTrainRatio
[float , optional] The desired ratio of the node data to use for training. The number must be between 0 and 1. The default is 0.8 which means 80% of the data will be used for training. This value is ignored if an nodeMaskKey is foud.

nodeValidateRatio
[float , optional] The desired ratio of the node data to use for validation. The number must be between 0 and 1. The default is 0.1 which means 10% of the data will be used for validation. This value is ignored if an nodeMaskKey is foud.

nodeTestRatio
[float , optional] The desired ratio of the node data to use for testing. The number must be between 0 and 1. The default is 0.1 which means 10% of the data will be used for testing. This value is ignored if an nodeMaskKey is foud.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

overwrite
[bool , optional] If set to True, any existing files are overwritten. Otherwise, the input list of graphs is appended to the end of each file. The default is False.

Returns

bool
True if the graph has been successfully exported. False otherwise.

`static ExportToGEXF`(*graph*, *path: str = None*, *graphWidth: float = 20*, *graphLength: float = 20*, *graphHeight: float = 20*, *defaultVertexColor: str = 'black'*, *defaultVertexSize: float = 3*, *vertexLabelKey: str = None*, *vertexColorKey: str = None*, *vertexSizeKey: str = None*, *defaultEdgeColor: str = 'black'*, *defaultEdgeWeight: float = 1*, *defaultEdgeType: str = 'undirected'*, *edgeLabelKey: str = None*, *edgeColorKey: str = None*, *edgeWeightKey: str = None*, *overwrite: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Exports the input graph to a Graph Exchange XML (GEXF) file format. See https://gexf.net/

Parameters

graph
[topologic_core.Graph] The input graph

path
[str] The desired path to the output folder where the graphs, edges, and nodes CSV files will be saved.

graphWidth
[float or int , optional] The desired graph width. The default is 20.

graphLength
[float or int , optional] The desired graph length. The default is 20.

graphHeight
[float or int , optional] The desired graph height. The default is 20.

defaultVertexColor
[str , optional] The desired default vertex color. The default is "black".

defaultVertexSize
[float or int , optional] The desired default vertex size. The default is 3.

defaultEdgeColor
[str , optional] The desired default edge color. The default is "black".

defaultEdgeWeight
[float or int , optional] The desired default edge weight. The edge weight determines the width of the displayed edge. The default is 3.

defaultEdgeType
[str , optional] The desired default edge type. This can be one of "directed" or "undirected". The default is "undirected".

vertexLabelKey
[str , optional] If specified, the vertex dictionary is searched for this key to determine the vertex label. If not specified the vertex label being is set to "Node X" where is X is a unique number. The default is None.

vertexColorKey
[str , optional] If specified, the vertex dictionary is searched for this key to determine the vertex color. If not specified the vertex color is set to the value defined by defaultVertex-Color parameter. The default is None.

vertexSizeKey
[str , optional] If specified, the vertex dictionary is searched for this key to determine the vertex size. If not specified the vertex size is set to the value defined by defaultVertexSize parameter. The default is None.

edgeLabelKey
[str , optional] If specified, the edge dictionary is searched for this key to determine the edge label. If not specified the edge label being is set to "Edge X" where is X is a unique number. The default is None.

edgeColorKey
[str , optional] If specified, the edge dictionary is searched for this key to determine the edge color. If not specified the edge color is set to the value defined by defaultEdgeColor parameter. The default is None.

edgeWeightKey
[str , optional] If specified, the edge dictionary is searched for this key to determine the edge

weight. If not specified the edge weight is set to the value defined by defaultEdgeWeight parameter. The default is None.

overwrite
[bool , optional] If set to True, any existing file is overwritten. Otherwise, it is not. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the graph has been successfully exported. False otherwise.

`static ExportToJSON`(*graph*, *path*, *verticesKey='vertices'*, *edgesKey='edges'*, *vertexLabelKey=''*, *edgeLabelKey=''*, *xKey='x'*, *yKey='y'*, *zKey='z'*, *indent=4*, *sortKeys=False*, *mantissa=6*, *overwrite=False*)

Exports the input graph to a JSON file.

Parameters

graph
[topologic_core.Graph] The input graph.

path
[str] The path to the JSON file.

verticesKey
[str , optional] The desired key name to call vertices. The default is "vertices".

edgesKey
[str , optional] The desired key name to call edges. The default is "edges".

vertexLabelKey
[str , optional] If set to a valid string, the vertex label will be set to the value at this key. Otherwise it will be set to Vertex_XXXX where XXXX is a sequential unique number. Note: If vertex labels are not unique, they will be forced to be unique.

edgeLabelKey
[str , optional] If set to a valid string, the edge label will be set to the value at this key. Otherwise it will be set to Edge_XXXX where XXXX is a sequential unique number. Note: If edge labels are not unique, they will be forced to be unique.

xKey
[str , optional] The desired key name to use for x-coordinates. The default is "x".

yKey
[str , optional] The desired key name to use for y-coordinates. The default is "y".

zKey
[str , optional] The desired key name to use for z-coordinates. The default is "z".

indent
[int , optional] The desired amount of indent spaces to use. The default is 4.

sortKeys
[bool , optional] If set to True, the keys will be sorted. Otherwise, they won't be. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

Returns

bool
The status of exporting the JSON file. If True, the operation was successful. Otherwise, it was unsuccesful.

`static GlobalClusteringCoefficient`(*graph*)

Returns the global clustering coefficient of the input graph. See https://en.wikipedia.org/wiki/Clustering_coefficient.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

int
The computed global clustering coefficient.

`static Guid`(*graph*)

Returns the guid of the input graph

Parameters

graph
[topologic_core.Graph] The input graph.

`static IncomingEdges`(*graph*, *vertex*, *directed: bool = False*, *tolerance: float = 0.0001*) → list

Returns the incoming edges connected to a vertex. An edge is considered incoming if its end vertex is coincident with the input vertex.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

directed
[bool , optional] If set to True, the graph is considered to be directed. Otherwise, it will be considered as an unidirected graph. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of incoming edges

`static IncomingVertices`(*graph*, *vertex*, *directed: bool = False*, *tolerance: float = 0.0001*) → list

Returns the incoming vertices connected to a vertex. A vertex is considered incoming if it is an adjacent vertex to the input vertex and the the edge connecting it to the input vertex is an incoming edge.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

directed
[bool , optional] If set to True, the graph is considered to be directed. Otherwise, it will be considered as an unidirected graph. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of incoming vertices

`static IsBipartite`(*graph*, *tolerance=0.0001*)

Returns True if the input graph is bipartite. Returns False otherwise. See https://en.wikipedia.org/wiki/Bipartite_graph.

Parameters

graph
[topologic_core.Graph] The input graph.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the input graph is complete. False otherwise

`static IsComplete`(*graph*)

Returns True if the input graph is complete. Returns False otherwise. See https://en.wikipedia.org/wiki/Complete_graph.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

bool
True if the input graph is complete. False otherwise

`static IsErdoesGallai`(*graph*, *sequence*)

Returns True if the input sequence satisfies the Erdős–Gallai theorem. Returns False otherwise. See https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Gallai_theorem.

Parameters

graph
[topologic_core.Graph] The input graph.

sequence
[list] The input sequence.

Returns

bool
True if the input sequence satisfies the Erdős–Gallai theorem. False otherwise.

`static IsTree`(*graph*)

Returns True if the input graph has a hierarchical tree-like structure. Returns False otherwise.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

bool
True if the input graph has a hierarchical tree-like structure. False otherwise.

`static IsolatedVertices`(*graph*)

Returns the list of isolated vertices in the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

list
The list of isolated vertices.

`static JSONData`(*graph*, *verticesKey: str = 'vertices'*, *edgesKey: str = 'edges'*, *vertexLabelKey: str = ''*, *edgeLabelKey: str = ''*, *sourceKey: str = 'source'*, *targetKey: str = 'target'*, *xKey: str = 'x'*, *yKey: str = 'y'*, *zKey: str = 'z'*, *geometryKey: str = 'brep'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Converts the input graph into JSON data.

Parameters

graph
[topologic_core.Graph] The input graph.

verticesKey
[str , optional] The desired key name to call vertices. The default is "vertices".

edgesKey
[str , optional] The desired key name to call edges. The default is "edges".

vertexLabelKey
[str , optional] If set to a valid string, the vertex label will be set to the value at this key. Otherwise it will be set to Vertex_XXXX where XXXX is a sequential unique number. Note: If vertex labels are not unique, they will be forced to be unique.

edgeLabelKey
[str , optional] If set to a valid string, the edge label will be set to the value at this key. Otherwise it will be set to Edge_XXXX where XXXX is a sequential unique number. Note: If edge labels are not unique, they will be forced to be unique.

sourceKey
[str , optional] The dictionary key used to store the source vertex. The default is "source".

targetKey
[str , optional] The dictionary key used to store the target vertex. The default is "target".

xKey
[str , optional] The desired key name to use for x-coordinates. The default is "x".

yKey
[str , optional] The desired key name to use for y-coordinates. The default is "y".

zKey
[str , optional] The desired key name to use for z-coordinates. The default is "z".

geometryKey
[str , optional] The desired key name to use for geometry. The default is "brep".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dict
The JSON data

`static JSONString`(*graph*, *verticesKey='vertices'*, *edgesKey='edges'*, *vertexLabelKey=''*, *edgeLabelKey=''*, *xKey='x'*, *yKey='y'*, *zKey='z'*, *indent=4*, *sortKeys=False*, *mantissa=6*)

Converts the input graph into JSON data.

Parameters

graph
[topologic_core.Graph] The input graph.

verticesKey
[str , optional] The desired key name to call vertices. The default is "vertices".

edgesKey
[str , optional] The desired key name to call edges. The default is "edges".

vertexLabelKey
[str , optional] If set to a valid string, the vertex label will be set to the value at this key. Otherwise it will be set to Vertex_XXXX where XXXX is a sequential unique number. Note: If vertex labels are not unique, they will be forced to be unique.

edgeLabelKey
[str , optional] If set to a valid string, the edge label will be set to the value at this key. Otherwise it will be set to Edge_XXXX where XXXX is a sequential unique number. Note: If edge labels are not unique, they will be forced to be unique.

xKey
[str , optional] The desired key name to use for x-coordinates. The default is "x".

yKey
[str , optional] The desired key name to use for y-coordinates. The default is "y".

zKey
[str , optional] The desired key name to use for z-coordinates. The default is "z".

indent
[int , optional] The desired amount of indent spaces to use. The default is 4.

sortKeys
[bool , optional] If set to True, the keys will be sorted. Otherwise, they won't be. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

str
The JSON str

`static LocalClusteringCoefficient`(*graph*, *vertices: list = None*, *key: str = 'lcc'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the local clustering coefficient of the input list of vertices within the input graph. See https://en.wikipedia.org/wiki/Clustering_coefficient.

Parameters

graph
[topologic_core.Graph] The input graph.

vertices
[list , optional] The input list of vertices. If set to None, the local clustering coefficient of all vertices will be computed. The default is None.

key
[str , optional] The dictionary key under which to save the local clustering coefficient score. The default is "lcc".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of local clustering coefficient. The order of the list matches the order of the list of input vertices.

`static LongestPath`(*graph*, *vertexA*, *vertexB*, *vertexKey=None*, *edgeKey=None*, *costKey=None*, *timeLimit=10*, *tolerance=0.0001*)

Returns the longest path that connects the input vertices.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

vertexKey
[str , optional] The vertex key to maximize. If set the vertices dictionaries will be searched for this key and the associated value will be used to compute the longest path that maximizes the total value. The value must be numeric. The default is None.

edgeKey
[str , optional] The edge key to maximize. If set the edges dictionaries will be searched for this key and the associated value will be used to compute the longest path that maximizes

the total value. The value of the key must be numeric. If set to "length" (case insensitive), the shortest path by length is computed. The default is "length".

costKey
[str , optional] If not None, the total cost of the longest_path will be stored in its dictionary under this key. The default is None.

timeLimit
[int , optional] The time limit in second. The default is 10 seconds.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The longest path between the input vertices.

`static MaximumDelta(graph)`

Returns the maximum delta of the input graph. The maximum delta of a graph is the maximum degree of a vertex in the graph.

Parameters

graph
[topologic_core.Graph] the input graph.

Returns

int
The maximum delta.

`static MaximumFlow(graph, source, sink, edgeKeyFwd=None, edgeKeyBwd=None, bidirKey=None, bidirectional=False, residualKey='residual', tolerance=0.0001)`

Returns the maximum flow of the input graph. See https://en.wikipedia.org/wiki/Maximum_flow_problem

Parameters

graph
[topologic_core.Graph] The input graph. This is assumed to be a directed graph

source
[topologic_core.Vertex] The input source vertex.

sink
[topologic_core.Vertex] The input sink/target vertex.

edgeKeyFwd
[str , optional] The edge dictionary key to use to find the value of the forward capacity of the edge. If not set, the length of the edge is used as its capacity. The default is None.

edgeKeyBwd
[str , optional] The edge dictionary key to use to find the value of the backward capacity of the edge. This is only considered if the edge is set to be bidirectional. The default is None.

bidirKey
[str , optional] The edge dictionary key to use to determine if the edge is bidirectional. The default is None.

bidirectional
[bool , optional] If set to True, the whole graph is considered to be bidirectional. The default is False.

residualKey
[str , optional] The name of the key to use to store the residual value of each edge capacity in the input graph. The default is "residual".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

float
The maximum flow.

`static MeshData`(*g*, *tolerance: float = 0.0001*)

Returns the mesh data of the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dict
The python dictionary of the mesh data of the input graph. The keys in the dictionary are: 'vertices' : The list of [x, y, z] coordinates of the vertices. 'edges' : the list of [i, j] indices into the vertices list to signify and edge that connects vertices[i] to vertices[j]. 'vertexDictionaries' : The python dictionaries of the vertices (in the same order as the list of vertices). 'edgeDictionaries' : The python dictionaries of the edges (in the same order as the list of edges).

`static MetricDistance`(*graph*, *vertexA*, *vertexB*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the shortest-path distance between the input vertices. See https://en.wikipedia.org/wiki/Distance_(graph_theory).

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

float
The shortest-path metric distance between the input vertices.

`static MinimumDelta`(*graph*)

Returns the minimum delta of the input graph. The minimum delta of a graph is the minimum degree of a vertex in the graph.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

int
The minimum delta.

`static MinimumSpanningTree`(*graph*, *edgeKey=None*, *tolerance=0.0001*)

Returns the minimum spanning tree of the input graph. See https://en.wikipedia.org/wiki/Minimum_spanning_tree.

Parameters

graph
[topologic_core.Graph] The input graph.

edgeKey
[string , optional] If set, the value of the edgeKey will be used as the weight and the tree will minimize the weight. The value associated with the edgeKey must be numerical. If the key is not set, the edges will be sorted by their length. The default is None

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The minimum spanning tree.

`static NavigationGraph`(*face*, *sources=None*, *destinations=None*, *tolerance=0.0001*, *numWorkers=None*)

Creates a 2D navigation graph.

Parameters

face
[topologic_core.Face] The input boundary. View edges will be clipped to this face. The holes in the face are used as the obstacles

sources
[list] The first input list of sources (vertices). Navigation edges will connect these veritces to destinations.

destinations
[list] The input list of destinations (vertices). Navigation edges will connect these vertices to sources.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

numWorkers
[int, optional] Number of workers run in parallel to process. The default is None which sets the number to twice the number of CPU cores.

Returns

topologic_core.Graph
The navigation graph.

`static NearestVertex`(*graph*, *vertex*)

Returns the vertex in the input graph that is the nearest to the input vertex.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

Returns

topologic_core.Vertex
The vertex in the input graph that is the nearest to the input vertex.

`static NetworkXGraph`(*graph*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

converts the input graph into a NetworkX Graph. See http://networkx.org

Parameters

graph
[topologic_core.Graph] The input graph.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

networkX Graph
The created networkX Graph

`static Order`(*graph*)

Returns the graph order of the input graph. The graph order is its number of vertices.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

int
The number of vertices in the input graph

`static OutgoingEdges`(*graph*, *vertex*, *directed: bool = False*, *tolerance: float = 0.0001*) → list

Returns the outgoing edges connected to a vertex. An edge is considered outgoing if its start vertex is coincident with the input vertex.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

directed
[bool , optional] If set to True, the graph is considered to be directed. Otherwise, it will be considered as an unidirected graph. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of outgoing edges

`static OutgoingVertices`(*graph*, *vertex*, *directed: bool = False*, *tolerance: float = 0.0001*) → list

Returns the list of outgoing vertices connected to a vertex. A vertex is considered outgoing if it is an adjacent vertex to the input vertex and the the edge connecting it to the input vertex is an outgoing edge.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

directed
[bool , optional] If set to True, the graph is considered to be directed. Otherwise, it will be considered as an unidrected graph. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of incoming vertices

`static PageRank`(*graph*, *alpha: float = 0.85*, *maxIterations: int = 100*, *normalize: bool = True*, *directed: bool = False*, *key: str = 'page_rank'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Calculates PageRank scores for nodes in a directed graph. see https://en.wikipedia.org/wiki/PageRank.

Parameters

graph
[topologic_core.Graph] The input graph.

alpha
[float , optional] The damping (dampening) factor. The default is 0.85. See https://en.wikipedia.org/wiki/PageRank.

maxIterations
[int , optional] The maximum number of iterations to calculate the page rank. The default is 100.

normalize
[bool , optional] If set to True, the results will be normalized from 0 to 1. Otherwise, they won't be. The default is True.

directed
[bool , optional] If set to True, the graph is considered as a directed graph. Otherwise, it will be considered as an undirected graph. The default is False.

key
[str , optional] The dictionary key under which to save the page_rank score. The default is "page_rank"

mantissa
[int , optional] The desired length of the mantissa.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of page ranks for the vertices in the graph.

`static Path`(*graph*, *vertexA*, *vertexB*, *tolerance=0.0001*)

Returns a path (wire) in the input graph that connects the input vertices.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The path (wire) in the input graph that connects the input vertices.

`static PyvisGraph`(*graph*, *path*, *overwrite: bool = True*, *height: int = 900*, *backgroundColor: str = 'white'*, *fontColor: str = 'black'*, *notebook: bool = False*, *vertexSize: int = 6*, *vertexSizeKey: str = None*, *vertexColor: str = 'black'*, *vertexColorKey: str = None*, *vertexLabelKey: str = None*, *vertexGroupKey: str = None*, *vertexGroups: list = None*, *minVertexGroup: float = None*, *maxVertexGroup: float = None*, *edgeLabelKey: str = None*, *edgeWeight: int = 0*, *edgeWeightKey: str = None*, *showNeighbours: bool = True*, *selectMenu: bool = True*, *filterMenu: bool = True*, *colorScale: str = 'viridis'*, *tolerance: float = 0.0001*)

Displays a pyvis graph. See https://pyvis.readthedocs.io/.

Parameters

graph
[topologic_core.Graph] The input graph.

path
[str] The desired file path to the HTML file into which to save the pyvis graph.

overwrite
[bool , optional] If set to True, the HTML file is overwritten.

height
[int , optional] The desired figure height in pixels. The default is 900 pixels.

backgroundColor
[str, optional] The desired background color for the figure. This can be a named color or a hexadecimal value. The default is 'white'.

fontColor
[str , optional] The desired font color for the figure. This can be a named color or a hexadecimal value. The default is 'black'.

notebook
[bool , optional] If set to True, the figure will be targeted at a Jupyter Notebook. Note that this is not working well. Pyvis has bugs. The default is False.

vertexSize
[int , optional] The desired default vertex size. The default is 6.

vertexSizeKey
[str , optional] If not set to None, the vertex size will be derived from the dictionary value set at this key. If set to "degree", the size of the vertex will be determined by its degree (number of neighbors). The default is None.

vertexColor
[str , optional] The desired default vertex color. his can be a named color or a hexadecimal value. The default is 'black'.

vertexColorKey
[str , optional] If not set to None, the vertex color will be derived from the dictionary value set at this key. The default is None.

vertexLabelKey
[str , optional] If not set to None, the vertex label will be derived from the dictionary value set at this key. The default is None.

vertexGroupKey
[str , optional] If not set to None, the vertex color will be determined by the group the vertex belongs to as derived from the value set at this key. The default is None.

vertexGroups
[list , optional] The list of all possible vertex groups. This will help in vertex coloring. The default is None.

minVertexGroup
[int or float , optional] If the vertex groups are numeric, specify the minimum value you wish to consider for vertex coloring. The default is None.

maxVertexGroup
[int or float , optional] If the vertex groups are numeric, specify the maximum value you wish to consider for vertex coloring. The default is None.

edgeWeight
[int , optional] The desired default weight of the edge. This determines its thickness. The default is 0.

edgeWeightKey
[str, optional] If not set to None, the edge weight will be derived from the dictionary value set at this key. If set to "length" or "distance", the weight of the edge will be determined by its geometric length. The default is None.

edgeLabelKey
[str , optional] If not set to None, the edge label will be derived from the dictionary value set at this key. The default is None.

showNeighbors
[bool , optional] If set to True, a list of neighbors is shown when you hover over a vertex. The default is True.

selectMenu
[bool , optional] If set to True, a selection menu will be displayed. The default is True

filterMenu
[bool , optional] If set to True, a filtering menu will be displayed. The default is True.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "viridis", "plasma"). The default is "viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns
——-

None
The pyvis graph is displayed either inline (notebook mode) or in a new browser window or tab.

`static RemoveEdge`(*graph*, *edge*, *tolerance=0.0001*)

Removes the input edge from the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

edge
[topologic_core.Edge] The input edge.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The input graph with the input edge removed.

`static RemoveVertex`(*graph*, *vertex*, *tolerance=0.0001*)

Removes the input vertex from the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The input graph with the input vertex removed.

`static Reshape`(*graph*, *shape='spring_2d'*, *k=0.8*, *seed=None*, *iterations=50*, *rootVertex=None*, *size=1*, *sides=16*, *key=''*, *tolerance=0.0001*, *silent=False*)

Reshapes the input graph according to the desired input shape parameter.

Parameters

graph
[topologic_core.Graph] The input graph.

shape
[str , optional] The desired shape of the graph. If set to 'spring_2d' or 'spring_3d', the algorithm uses a simplified version of the Fruchterman-Reingold force-directed algorithm to distribute the vertices. If set to 'radial_2d', the nodes will be distributed along concentric circles in the XY plane. If set to 'tree_2d' or 'tree_3d', the nodes will be distributed using the Reingold-Tillford layout. If set to 'circle_2d', the nodes will be distributed on the cirumference of a segemented circles in the XY plane, based on the size and sides input parameter (radius=size/2). If set to 'line_2d', the nodes will be distributed on a line in the XY plane based on the size input parameter (length=size). If set to 'spehere_3d', the nodes will be distributed on the surface of a sphere based on the size input parameter raidus=size/2). If set to 'grid_2d', the nodes will be distributed on a grid in the XY plane with size based on the size input parameter (length=width=size). If set to 'grid_3d', the nodes will be distributed on a 3D cubic grid/matrix based on the size input parameter(width=length=height=size). If set to 'cluster_2d', or 'cluster_3d, the nodes will be clustered according to the 'key' input parameter. The overall radius of the cluster is determined by the size input parameter (radius = size/2) The default is 'spring_2d'.

k
[float, optional] The desired spring constant to use for the attractive and repulsive forces. The default is 0.8.

seed
[int , optional] The desired random seed to use. The default is None.

iterations
[int , optional] The desired maximum number of iterations to solve the forces in the 'spring' mode. The default is 50.

rootVertex
[topologic_core.Vertex , optional] The desired vertex to use as the root of the tree and radial layouts.

sides
[int , optional] The desired number of sides of the circle layout option. The default is 16

length
[float, optional] The desired horizontal length for the line layout option. The default is 1.0.

key
[string, optional] The key under which to find the clustering value for the 'cluster_2d' and 'cluster_3d' options. The default is "".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Graph
The reshaped graph.

`static SetDictionary`(*graph*, *dictionary*)

Sets the input graph's dictionary to the input dictionary

Parameters

graph
[topologic_core.Graph] The input graph.

dictionary
[topologic_core.Dictionary or dict] The input dictionary.

Returns

topologic_core.Graph
The input graph with the input dictionary set in it.

`static ShortestPath`(*graph*, *vertexA*, *vertexB*, *vertexKey=''*, *edgeKey='Length'*, *tolerance=0.0001*)

Returns the shortest path that connects the input vertices. The shortest path will take into consideration both the vertexKey and the edgeKey if both are specified and will minimize the total "cost" of the path. Otherwise, it will take into consideration only whatever key is specified.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

vertexKey
[string , optional] The vertex key to minimise. If set the vertices dictionaries will be searched for this key and the associated value will be used to compute the shortest path that minimized the total value. The value must be numeric. The default is None.

edgeKey
[string , optional] The edge key to minimise. If set the edges dictionaries will be searched for this key and the associated value will be used to compute the shortest path that minimized the total value. The value of the key must be numeric. If set to "length" (case insensitive), the shortest path by length is computed. The default is "length".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The shortest path between the input vertices.

`static ShortestPaths`(*graph*, *vertexA*, *vertexB*, *vertexKey=''*, *edgeKey='length'*, *timeLimit=10*, *pathLimit=10*, *tolerance=0.0001*)

Returns the shortest path that connects the input vertices.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

vertexKey
[string , optional] The vertex key to minimise. If set the vertices dictionaries will be searched for this key and the associated value will be used to compute the shortest path that minimized the total value. The value must be numeric. The default is None.

edgeKey
[string , optional] The edge key to minimise. If set the edges dictionaries will be searched for this key and the associated value will be used to compute the shortest path that minimized the total value. The value of the key must be numeric. If set to "length" (case insensitive), the shortest path by length is computed. The default is "length".

timeLimit
[int , optional] The search time limit in seconds. The default is 10 seconds

pathLimit: int , optional
The number of found paths limit. The default is 10 paths.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of shortest paths between the input vertices.

`static Show`(*graph*, *sagitta=0*, *absolute=False*, *sides=8*, *angle=0*, *vertexColor='black'*, *vertexColorKey=None*, *vertexSize=6*, *vertexSizeKey=None*, *vertexLabelKey=None*, *vertexGroupKey=None*, *vertexGroups=[]*, *showVertices=True*, *showVertexLabel=False*, *showVertexLegend=False*, *edgeColor='black'*, *edgeColorKey=None*, *edgeWidth=1*, *edgeWidthKey=None*, *edgeLabelKey=None*, *edgeGroupKey=None*, *edgeGroups=[]*, *showEdges=True*, *showEdgeLabel=False*, *showEdgeLegend=False*, *colorScale='viridis'*, *renderer=None*, *width=950*, *height=500*, *xAxis=False*, *yAxis=False*, *zAxis=False*, *axisSize=1*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=20*, *marginBottom=0*, *camera=[-1.25, -1.25, 1.25]*, *center=[0, 0, 0]*, *up=[0, 0, 1]*, *projection='perspective'*, *tolerance=0.0001*, *silent=False*)

Shows the graph using Plotly.

Parameters

graph
[topologic_core.Graph] The input graph.

sagitta
[float , optional] The length of the sagitta. In mathematics, the sagitta is the line connecting the center of a chord to the apex (or highest point) of the arc subtended by that chord. The default is 0 which means a straight edge is drawn instead of an arc. The default is 0.

absolute
[bool , optional] If set to True, the sagitta length is treated as an absolute value. Otherwise, it is treated as a ratio based on the length of the edge. The default is False. For example, if the length of the edge is 10, the sagitta is set to 0.5, and absolute is set to False, the sagitta length will be 5. The default is True.

sides
[int , optional] The number of sides of the arc. The default is 8.

angle
[float, optional] An additional angle in degrees to rotate arcs (where sagitta is more than 0). The default is 0.

vertexColor
[str , optional] The desired color of the output vertices. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

vertexColorKey
[str , optional] The dictionary key under which to find the vertex color. The default is None.

vertexSize
[float , optional] The desired size of the vertices. The default is 1.1.

vertexSizeKey
[str , optional] The dictionary key under which to find the vertex size. The default is None.

vertexLabelKey
[str , optional] The dictionary key to use to display the vertex label. The default is None.

vertexGroupKey
[str , optional] The dictionary key to use to display the vertex group. The default is None.

vertexGroups
[list , optional] The list of vertex groups against which to index the color of the vertex. The default is [].

showVertices
[bool , optional] If set to True the vertices will be drawn. Otherwise, they will not be drawn. The default is True.

showVertexLabel
[bool , optional] If set to True, the vertex labels are shown permenantely on screen. Otherwise, they are not. The default is False.

showVertexLegend
[bool , optional] If set to True the vertex legend will be drawn. Otherwise, it will not be drawn. The default is False.

edgeColor
[str , optional] The desired color of the output edges. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

edgeColorKey
[str , optional] The dictionary key under which to find the edge color. The default is None.

edgeWidth
[float , optional] The desired thickness of the output edges. The default is 1.

edgeWidthKey
[str , optional] The dictionary key under which to find the edge width. The default is None.

edgeLabelKey
[str , optional] The dictionary key to use to display the edge label. The default is None.

edgeGroupKey
[str , optional] The dictionary key to use to display the edge group. The default is None.

edgeGroups
[list , optional] The list of edge groups against which to index the color of the edge. The default is [].

showEdges
[bool , optional] If set to True the edges will be drawn. Otherwise, they will not be drawn. The default is True.

showEdgeLabel
[bool , optional] If set to True, the edge labels are shown permenantely on screen. Otherwise, they are not. The default is False.

showEdgeLegend
[bool , optional] If set to True the edge legend will be drawn. Otherwise, it will not be drawn. The default is False.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "Viridis", "Plasma"). The default is "Viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

renderer
[str , optional] The desired renderer. See Plotly.Renderers(). If set to None, the code will attempt to discover the most suitable renderer. The default is None.

width
[int , optional] The width in pixels of the figure. The default value is 950.

height
[int , optional] The height in pixels of the figure. The default value is 950.

xAxis
[bool , optional] If set to True the x axis is drawn. Otherwise it is not drawn. The default is False.

yAxis
[bool , optional] If set to True the y axis is drawn. Otherwise it is not drawn. The default is False.

zAxis
[bool , optional] If set to True the z axis is drawn. Otherwise it is not drawn. The default is False.

axisSize
[float , optional] The size of the X, Y, Z, axes. The default is 1.

backgroundColor
[str , optional] The desired color of the background. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "rgba(0,0,0,0)".

marginLeft
[int , optional] The size in pixels of the left margin. The default value is 0.

marginRight
[int , optional] The size in pixels of the right margin. The default value is 0.

marginTop
[int , optional] The size in pixels of the top margin. The default value is 20.

marginBottom
[int , optional] The size in pixels of the bottom margin. The default value is 0.

camera
[list , optional] The desired location of the camera). The default is [-1.25, -1.25, 1.25].

center
[list , optional] The desired center (camera target). The default is [0, 0, 0].

up
[list , optional] The desired up vector. The default is [0, 0, 1].

projection
[str , optional] The desired type of projection. The options are "orthographic" or "perspective". It is case insensitive. The default is "perspective"

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

None

`static Size`(*graph*)

Returns the graph size of the input graph. The graph size is its number of edges.

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

int
The number of edges in the input graph.

`static TopologicalDistance`(*graph*, *vertexA*, *vertexB*, *tolerance=0.0001*)

Returns the topological distance between the input vertices. See https://en.wikipedia.org/wiki/Distance_(graph_theory).

Parameters

graph
[topologic_core.Graph] The input graph.

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

int
The topological distance between the input vertices.

`static Topology`(*graph*)

Returns the topology (cluster) of the input graph

Parameters

graph
[topologic_core.Graph] The input graph.

Returns

topologic_core.Cluster
The topology of the input graph.

`static Tree`(*graph*, *vertex=None*, *tolerance=0.0001*)

Creates a tree graph version of the input graph rooted at the input vertex.

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex , optional] The input root vertex. If not set, the first vertex in the graph is set as the root vertex. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The tree graph version of the input graph.

`static VertexDegree`(*graph*, *vertex*, *edgeKey: str = None*, *tolerance: float = 0.0001*)

Returns the degree of the input vertex. See https://en.wikipedia.org/wiki/Degree_(graph_theory).

Parameters

graph
[topologic_core.Graph] The input graph.

vertex
[topologic_core.Vertex] The input vertex.

edgeKey
[str , optional] If specified, the value in the connected edges' dictionary specified by the edgeKey string will be aggregated to calculate the vertex degree. If a numeric value cannot be retrieved from an edge, a value of 1 is used instead. This is used in weighted graphs.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

int
The degree of the input vertex.

`static Vertices`(*graph*, *vertexKey=None*, *reverse=False*)

Returns the list of vertices in the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

vertexKey
[str , optional] If set, the returned list of vertices is sorted according to the dicitonary values stored under this key. The default is None.

reverse
[bool , optional] If set to True, the vertices are sorted in reverse order (only if vertexKey is set). Otherwise, they are not. The default is False.

Returns

list
The list of vertices in the input graph.

`static VisibilityGraph`(*face*, *viewpointsA=None*, *viewpointsB=None*, *tolerance=0.0001*)

Creates a 2D visibility graph.

Parameters

face
[topologic_core.Face] The input boundary. View edges will be clipped to this face. The holes in the face are used as the obstacles

viewpointsA
[list , optional] The first input list of viewpoints (vertices). Visibility edges will connect these veritces to viewpointsB. If set to None, this parameters will be set to all vertices of the input face. The default is None.

viewpointsB
[list , optional] The input list of viewpoints (vertices). Visibility edges will connect these vertices to viewpointsA. If set to None, this parameters will be set to all vertices of the input face. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The visibility graph.

`class topologicpy.Graph.GraphQueueItem`(*edges*)

Bases: `tuple`

Attributes

edges
Alias for field number 0

Methods

`count`(value, /)	Return number of occurrences of value.
`index`(value[, start, stop])	Return first index of value.

edges
Alias for field number 0

`class topologicpy.Graph.MergingProcess`(*message_queue*)

Bases: `Process`

Receive message from other processes and merging the result

Attributes

authkey
daemon
Return whether process is a daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

name
pid
Return identifier (PID) of process or *None* if it has yet to start

sentinel
Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.

Methods

close()	Close the Process object.
is_alive()	Return whether process is alive
join([timeout])	Wait until child process terminates
kill()	Terminate process; sends SIGKILL signal or uses TerminateProcess()
run()	Method to be run in sub-process; can be overridden in sub-class
start()	Start child process
terminate()	Terminate process; sends SIGTERM signal or uses TerminateProcess()

wait_message	

wait_message()

class topologicpy.Graph.**WorkerProcess**(*start_index*, *message_queue*, *used*, *face*, *sources*, *destinations*, *tolerance=0.0001*)

Bases: Process

Creates a 2D navigation graph from a subset of sources and the list of destinations.

Attributes

authkey
daemon
Return whether process is a daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

name
pid
Return identifier (PID) of process or *None* if it has yet to start

sentinel
Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.

Methods

close()	Close the Process object.
is_alive()	Return whether process is alive
join([timeout])	Wait until child process terminates
kill()	Terminate process; sends SIGKILL signal or uses TerminateProcess()
run()	Method to be run in sub-process; can be overridden in sub-class
start()	Start child process
terminate()	Terminate process; sends SIGTERM signal or uses TerminateProcess()

run()

Method to be run in sub-process; can be overridden in sub-class

class topologicpy.Graph.**WorkerProcessPool**(*num_workers*, *message_queue*, *used*, *face*, *sources*, *destinations*, *tolerance=0.0001*)

Bases: object

Create and manage a list of Worker processes. Each worker process creates a 2D navigation graph.

Methods

join	
startProcesses	
stopProcesses	

join()

startProcesses()

stopProcesses()

topologicpy.Grid module

class topologicpy.Grid.**Grid**

Bases: object

Methods

EdgesByDistances([face, uOrigin, vOrigin, ...])	Creates a grid (cluster of edges).
EdgesByParameters(face[, uRange, vRange, ...])	Creates a grid (cluster of edges).
VerticesByDistances([face, origin, uRange, ...])	Creates a grid (cluster of vertices).
VerticesByParameters([face, uRange, vRange, ...])	Creates a grid (cluster of vertices).

static EdgesByDistances(*face=None*, *uOrigin=None*, *vOrigin=None*, *uRange=[-0.5, -0.25, 0, 0.25, 0.5]*, *vRange=[-0.5, -0.25, 0, 0.25, 0.5]*, *clip=False*, *mantissa: int = 6*, *tolerance=0.0001*)

Creates a grid (cluster of edges).

Parameters

face
[topologic_core.Face , optional] The input face. If set to None, the grid will be created on the XY plane. The default is None.

uOrigin
[topologic_core.Vertex , optional] The origin of the *u* grid lines. If set to None: if the face is set, the uOrigin will be set to vertex at the face's 0,0 paratmer. If the face is set to None, the uOrigin will be set to the origin. The default is None.

vOrigin
[topologic_core.Vertex , optional] The origin of the *v* grid lines. If set to None: if the face is set, the vOrigin will be set to vertex at the face's 0,0 paratmer. If the face is set to None, the vOrigin will be set to the origin. The default is None.

uRange
[list , optional] A list of distances for the *u* grid lines from the uOrigin. The default is [-0.5,-0.25,0, 0.25,0.5].

vRange
[list , optional] A list of distances for the *v* grid lines from the vOrigin. The default is [-0.5,-0.25,0, 0.25,0.5].

clip
[bool , optional] If True the grid will be clipped by the shape of the input face. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The created grid. Edges in the grid have an identifying dictionary with two keys: "dir" and "offset". The "dir" key can have one of two values: "u" or "v", the "offset" key contains the offset distance of that grid edge from the specified origin.

static EdgesByParameters(*face*, *uRange=[0, 0.25, 0.5, 0.75, 1.0]*, *vRange=[0, 0.25, 0.5, 0.75, 1.0]*, *clip=False*, *tolerance=0.0001*)

Creates a grid (cluster of edges).

Parameters

face
[topologic_core.Face] The input face.

uRange
[list , optional] A list of *u* parameters for the *u* grid lines. The default is [0,0.25,0.5, 0.75, 1.0].

vRange
[list , optional] A list of *v* parameters for the *v* grid lines. The default is [0,0.25,0.5, 0.75, 1.0].

clip
[bool , optional] If True the grid will be clipped by the shape of the input face. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The created grid. Edges in the grid have an identifying dictionary with two keys: "dir" and "offset". The "dir" key can have one of two values: "u" or "v", the "offset" key contains the offset parameter of that grid edge.

`static VerticesByDistances`(*face=None*, *origin=None*, *uRange: list = [-0.5, -0.25, 0, 0.25, 0.5]*, *vRange: list = [-0.5, -0.25, 0, 0.25, 0.5]*, *clip: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a grid (cluster of vertices).

Parameters

face
[topologic_core.Face , optional] The input face. If set to None, the grid will be created on the XY plane. The default is None.

origin
[topologic_core.Vertex , optional] The origin of the grid vertices. If set to None: if the face is set, the origin will be set to vertex at the face's 0,0 paratmer. If the face is set to None, the origin will be set to (0, 0, 0). The default is None.

uRange
[list , optional] A list of distances for the *u* grid lines from the uOrigin. The default is [-0.5,-0.25,0, 0.25,0.5].

vRange
[list , optional] A list of distances for the *v* grid lines from the vOrigin. The default is [-0.5,-0.25,0, 0.25,0.5].

clip
[bool , optional] If True the grid will be clipped by the shape of the input face. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The created grid. Vertices in the grid have an identifying dictionary with two keys: "u" and "v". The "dir" key can have one of two values: "u" or "v" that contain the *u* and *v* offset distances of that grid vertex from the specified origin.

`static VerticesByParameters`(*face=None*, *uRange=[0.0, 0.25, 0.5, 0.75, 1.0]*, *vRange=[0.0, 0.25, 0.5, 0.75, 1.0]*, *clip=False*, *tolerance=0.0001*)

Creates a grid (cluster of vertices).

Parameters

face
[topologic_core.Face , optional] The input face. If set to None, the grid will be created on the XY plane. The default is None.

uRange
[list , optional] A list of *u* parameters for the *u* grid lines from the uOrigin. The default is [0.0,0.25,0.5,0.75,1.0].

vRange
[list , optional] A list of *v* parameters for the *v* grid lines from the vOrigin. The default is [0.0,0.25,0.5,0.75,1.0].

clip
[bool , optional] If True the grid will be clipped by the shape of the input face. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The created grid. Vertices in the grid have an identifying dictionary with two keys: "u" and "v". The "dir" key can have one of two values: "u" or "v" that contain the *u* and *v* offset distances of that grid vertex from the specified origin.

topologicpy.Helper module

`class topologicpy.Helper.Helper`

Bases: `object`

Methods

ClosestMatch(item, listA)	Returns the index of the closest match in the input list to the input item.
ClusterByKeys(elements, dictionaries, *keys)	Clusters the input list of elements and dictionaries based on the input key or keys.
Flatten(listA)	Flattens the input nested list.
Iterate(listA)	Iterates the input nested list so that each sublist has the same number of members.
MakeUnique(listA)	Forces the strings in the input list to be unique if they have duplicates.
MergeByThreshold(listA[, threshold])	Merges the numbers in the input list so that numbers within the input threshold are averaged into one number.
Normalize(listA[, mantissa])	Normalizes the input list so that it is in the range 0 to 1
Position(item, listA)	Returns the position of the item in the list or the position it would have been inserts.
Repeat(listA)	Repeats the input nested list so that each sublist has the same number of members.
Sort(listA, *otherLists[, reverseFlags])	Sorts the first input list according to the values in the subsequent input lists in order.
Transpose(listA)	Transposes the input list (swaps rows and columns).
Trim(listA)	Trims the input nested list so that each sublist has the same number of members.
Version()	Returns the current version of the software.

static ClosestMatch(*item*, *listA*)

Returns the index of the closest match in the input list to the input item. This works for lists made out of numeric or string values.

Parameters

item
[int, float, or str] The input item.

listA
[list] The input list.

Returns

int
The index of the best match in listA for the input item.

static ClusterByKeys(*elements*, *dictionaries*, **keys*, *silent=False*)

Clusters the input list of elements and dictionaries based on the input key or keys.

Parameters

elements
[list] The input list of elements to be clustered.

dictionaries
[list[Topology.Dictionary]] The input list of dictionaries to be consulted for clustering. This is assumed to be in the same order as the list of elements.

keys
[str or list or comma-separated str input parameters] The key or keys in the topology's dictionary to use for clustering.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

dict
A dictionary containing the elements and the dictionaries, but clustered. The dictionary has two keys: "elements": list

A nested list of elements where each item is a list of elements with the same key values.

"dictionaries": list
A nested list of dictionaries where each item is a list of dictionaries with the same key values.

`static Flatten`(*listA*)

Flattens the input nested list.

Parameters

listA
[list] The input nested list.

Returns

list
The flattened list.

`static Iterate`(*listA*)

Iterates the input nested list so that each sublist has the same number of members. To fill extra members, the shorter lists are iterated from their first member. For example Iterate([[1,2,3],['m','n','o','p'],['a','b','c','d','e']]) yields [[1, 2, 3, 1, 2], ['m', 'n', 'o', 'p', 'm'], ['a', 'b', 'c', 'd', 'e']]

Parameters

listA
[list] The input nested list.

Returns

list
The iterated list.

`static MakeUnique`(*listA*)

Forces the strings in the input list to be unique if they have duplicates.

Parameters

listA
[list] The input list of strings.

Returns

list
The input list, but with each item ensured to be unique if they have duplicates.

static MergeByThreshold(*listA*, *threshold=0.0001*)

Merges the numbers in the input list so that numbers within the input threshold are averaged into one number.

Parameters

listA
[list] The input nested list.

threshold
[float , optional] The desired merge threshold value. The default is 0.0001.

Returns

list
The merged list. The list is sorted in ascending numeric order.

static Normalize(*listA*, *mantissa: int = 6*)

Normalizes the input list so that it is in the range 0 to 1

Parameters

listA
[list] The input nested list.

mantissa
[int , optional] The desired mantissa value. The default is 6.

Returns

list
The normalized list.

static Position(*item*, *listA*)

Returns the position of the item in the list or the position it would have been inserts. item is assumed to be numeric. listA is assumed to contain only numeric values and sorted from lowest to highest value.

Parameters

item
[int or float] The input number to be positioned.

listA
[list] The input sorted list.

Returns

int
The position of the item within the list.

static Repeat(*listA*)

Repeats the input nested list so that each sublist has the same number of members. To fill extra members, the last item in the shorter lists are repeated and appended. For example Iterate([[1,2,3],['m','n','o','p'],['a','b','c','d','e']]) yields [[1, 2, 3, 3, 3], ['m', 'n', 'o', 'p', 'p'], ['a', 'b', 'c', 'd', 'e']]

Parameters

listA
[list] The input nested list.

Returns

list
The repeated list.

`static Sort`(*listA*, **otherLists*, *reverseFlags=None*)

Sorts the first input list according to the values in the subsequent input lists in order. For example, your first list can be a list of topologies and the next set of lists can be their volume, surface area, and z level. The list of topologies will then be sorted first by volume, then by surface, and lastly by z level. You can choose to reverse the order of sorting by including a list of TRUE/FALSE values in the reverseFlags input parameter. For example, if you wish to sort the volume in reverse order (from large to small), but sort the other parameters normally, you would include the following list for reverseFlag: [True, False, False].

Parameters

listA
[list] The first input list to be sorts

***otherLists**
[any number of lists to use for sorting listA, optional.] Any number of lists that are used to sort the listA input parameter. The order of these input parameters determines the order of sorting (from left to right). If no lists are included, the input list will be sorted as is.

reverseFlags
[list, optional.] The list of booleans (TRUE/FALSE) to indicated if sorting based on a particular list should be conducted in reverse order. The length of the reverseFlags list should match the number of the lists in the input otherLists parameter. If set to None, a default list of FALSE values is created to match the number of the lists in the input otherLists parameter. The default is None.

Returns

list
The sorted list.

`static Transpose`(*listA*)

Transposes the input list (swaps rows and columns).

Parameters

listA
[list] The input list.

Returns

list
The transposed list.

`static Trim`(*listA*)

Trims the input nested list so that each sublist has the same number of members. All lists are trimmed to match the length of the shortest list. For example Trim([[1,2,3],[‘m’,’n’,’o’,’p’],[‘a’,’b’,’c’,’d’,’e’]]) yields [[1, 2, 3], [‘m’, ‘n’, ‘o’], [‘a’, ‘b’, ‘c’]]

Parameters

listA
[list] The input nested list.

Returns

list
The repeated list.

static Version()

Returns the current version of the software.

Parameters

Returns

str
The current version of the software.

topologicpy.Honeybee module

class topologicpy.Honeybee.Honeybee

Bases: object

Methods

ConstructionSetByIdentifier(id)	Returns the built-in construction set by the input identifying string.
ConstructionSets()	Returns the list of built-in construction sets
ExportToHBJSON(model, path[, overwrite])	Exports the input HB Model to a file.
ModelByTopology(tpBuilding[, ...])	Creates an HB Model from the input Topology.
ProgramTypeByIdentifier(id)	Returns the program type by the input identifying string.
ProgramTypes()	Returns the list of available built-in program types.
String(model)	Returns the string representation of the input model.

static ConstructionSetByIdentifier(*id*)

Returns the built-in construction set by the input identifying string.

Parameters

id
[str] The construction set identifier.

Returns

HBConstructionSet
The found built-in construction set.

static ConstructionSets()

Returns the list of built-in construction sets

Returns

list
The list of built-in construction sets.

static ExportToHBJSON(*model*, *path*, *overwrite=False*)

Exports the input HB Model to a file.

Parameters

model
[HBModel] The input HB Model.

path
[str] The location of the output file.

overwrite
[bool , optional] If set to True this method overwrites any existing file. Otherwise, it won't. The default is False.

Returns

bool
Returns True if the operation is successful. Returns False otherwise.

`static ModelByTopology`(*tpBuilding*, *tpShadingFacesCluster=None*, *buildingName: str = 'Generic_Building'*, *defaultProgramIdentifier: str = 'Generic Office Program'*, *defaultConstructionSetIdentifier: str = 'Default Generic Construction Set'*, *coolingSetpoint: float = 25.0*, *heatingSetpoint: float = 20.0*, *humidifyingSetpoint: float = 30.0*, *dehumidifyingSetpoint: float = 55.0*, *roomNameKey: str = 'TOPOLOGIC_name'*, *roomTypeKey: str = 'TOPOLOGIC_type'*, *apertureTypeKey: str = 'TOPOLOGIC_type'*, *addSensorGrid: bool = False*, *mantissa: int = 6*)

Creates an HB Model from the input Topology.

Parameters

tpBuilding
[topologic_core.CellComplex or topologic_core.Cell] The input building topology.

tpShadingFaceCluster
[topologic_core.Cluster , optional] The input cluster for shading faces. The default is None.

buildingName
[str , optional] The desired name of the building. The default is "Generic_Building".

defaultProgramIdentifier: str , optional
The desired default program identifier. The default is "Generic Office Program".

defaultConstructionSetIdentifier: str , optional
The desired default construction set identifier. The default is "Default Generic Construction Set".

coolingSetpoint
[float , optional] The desired HVAC cooling set point in degrees Celsius. The default is 25.

heatingSetpoint
[float , optional] The desired HVAC heating set point in degrees Celsius. The default is 20.

humidifyingSetpoint
[float , optional] The desired HVAC humidifying set point in percentage. The default is 55.

roomNameKey
[str , optional] The dictionary key under which the room name is stored. The default is "TOPOLOGIC_name".

roomTypeKey
[str , optional] The dictionary key under which the room type is stored. The default is "TOPOLOGIC_type".

apertureTypeKey
[str , optional] The dictionary key under which the aperture type is stored. The default is "TOPOLOGIC_type".

addSensorGrid
[bool , optional] If set to True a sensor grid is add to horizontal surfaces. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

HBModel
The created HB Model

`static ProgramTypeByIdentifier(id)`

Returns the program type by the input identifying string.

Parameters

id
[str] The identifiying string.

Returns

HBProgram
The found built-in program.

`static ProgramTypes()`

Returns the list of available built-in program types.

Returns

list
The list of available built-in program types.

`static String(model)`

Returns the string representation of the input model.

Parameters

model
[HBModel] The input HB Model.

Returns

dict
A dictionary representing the input HB Model.

topologicpy.Matrix module

`class topologicpy.Matrix.Matrix`

Bases: `object`

Methods

Add(matA, matB)	Adds the two input matrices.
ByRotation([angleX, angleY, angleZ, order])	Creates a 4x4 rotation matrix.
ByScaling([scaleX, scaleY, scaleZ])	Creates a 4x4 scaling matrix.
ByTranslation([translateX, translateY, ...])	Creates a 4x4 translation matrix.
Multiply(matA, matB)	Multiplies the two input matrices.
Subtract(matA, matB)	Subtracts the two input matrices.
Transpose(matrix)	Transposes the input matrix.

static Add(*matA*, *matB*)

Adds the two input matrices.

Parameters

matA
[list] The first input matrix.

matB
[list] The second input matrix.

Returns

list
The matrix resulting from the addition of the two input matrices.

static ByRotation(*angleX=0*, *angleY=0*, *angleZ=0*, *order='xyz'*)

Creates a 4x4 rotation matrix.

Parameters

angleX
[float , optional] The desired rotation angle in degrees around the X axis. The default is 0.

angleY
[float , optional] The desired rotation angle in degrees around the Y axis. The default is 0.

angleZ
[float , optional] The desired rotation angle in degrees around the Z axis. The default is 0.

order
[string , optional] The order by which the roatations will be applied. The possible values are any permutation of "xyz". This input is case insensitive. The default is "xyz".

Returns

list
The created 4X4 rotation matrix.

static ByScaling(*scaleX=1.0*, *scaleY=1.0*, *scaleZ=1.0*)

Creates a 4x4 scaling matrix.

Parameters

scaleX
[float , optional] The desired scaling factor along the X axis. The default is 1.

scaleY
[float , optional] The desired scaling factor along the X axis. The default is 1.

scaleZ
[float , optional] The desired scaling factor along the X axis. The default is 1.

Returns

list
The created 4X4 scaling matrix.

`static ByTranslation`(*translateX=0*, *translateY=0*, *translateZ=0*)

Creates a 4x4 translation matrix.

Parameters

translateX
[float , optional] The desired translation distance along the X axis. The default is 0.

translateY
[float , optional] The desired translation distance along the X axis. The default is 0.

translateZ
[float , optional] The desired translation distance along the X axis. The default is 0.

Returns

list
The created 4X4 translation matrix.

`static Multiply`(*matA*, *matB*)

Multiplies the two input matrices. When transforming an object, the first input matrix is applied first then the second input matrix.

Parameters

matA
[list] The first input matrix.

matB
[list] The second input matrix.

Returns

list
The matrix resulting from the multiplication of the two input matrices.

`static Subtract`(*matA*, *matB*)

Subtracts the two input matrices.

Parameters

matA
[list] The first input matrix.

matB
[list] The second input matrix.

Returns

list
The matrix resulting from the subtraction of the second input matrix from the first input matrix.

`static Transpose`(*matrix*)

Transposes the input matrix.

Parameters

matrix
[list] The input matrix.

Returns

list
The transposed matrix.

topologicpy.Neo4j module

class topologicpy.Neo4j.**Neo4j**

Bases: object

Methods

ByGraph(graph[, vertexLabelKey, ...])	Converts a Topologic graph to a Neo4j graph.
ByParameters(url, username, password)	Returns a Neo4j graph by the input parameters.
ExportToGraph(neo4jGraph[, cypher, xMin, ...])	Exports the input neo4j graph to a topologic graph.
Reset(neo4jGraph)	Resets the database completely.
SetGraph(neo4jGraph, graph[, labelKey, ...])	Sets the input topologic graph to the input neo4jGraph.

ByGraph(*graph*, *vertexLabelKey: str = 'label'*, *defaultVertexLabel: str = 'NODE'*, *vertexCategoryKey: str = 'category'*, *defaultVertexCategory: str = None*, *edgeLabelKey: str = 'label'*, *defaultEdgeLabel: str = 'CONNECTED_TO'*, *edgeCategoryKey: str = 'category'*, *defaultEdgeCategory: str = None*, *bidirectional: bool = True*, *mantissa: int = 6*, *tolerance: float = 0.0001*, *silent: bool = False*)

Converts a Topologic graph to a Neo4j graph.

Parameters

neo4jGraph
[neo4j._sync.driver.BoltDriver or neo4jGraph, neo4j._sync.driver.Neo4jDriver] The input neo4j driver.

vertexLabelKey
[str , optional] The returned vertices are labelled according to the dictionary values stored under this key. If the vertexLabelKey does not exist, it will be created and the vertices are labelled numerically using the format defaultVertexLabel_XXX. The default is "label".

defaultVertexLabel
[str , optional] The default vertex label to use if no value is found under the vertexLabelKey. The default is "NODE".

vertexCategoryKey
[str , optional] The returned vertices are categorized according to the dictionary values stored under this key. The dfefault is "category".

defaultVertexCategory
[str , optional] The default vertex category to use if no value is found under the vertexCategoryKey. The default is None.

edgeLabelKey
[str , optional] The returned edges are labelled according to the dictionary values stored

under this key. If the edgeLabelKey does not exist, it will be created and the edges are labelled numerically using the format defaultEdgeLabel_XXX. The default is "label".

defaultEdgeLabel
[str , optional] The default edge label to use if no value is found under the edgeLabelKey. The default is "CONNECTED_TO".

edgeCategoryKey
[str , optional] The returned edges are categorized according to the dictionary values stored under this key. The dfefault is "category".

defaultEdgeCategory
[str , optional] The default edge category to use if no value is found under the edgeCategoryKey. The default is None.

bidirectional
[bool , optional] If set to True, the output Neo4j graph is forced to be bidirectional. The defaul is True.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

neo4j._sync.driver.BoltDriver or neo4jGraph, neo4j._sync.driver.Neo4jDriver
The returned neo4j driver.

`static ByParameters`(*url*, *username*, *password*)

Returns a Neo4j graph by the input parameters.

Parameters

url
[str] The URL of the server.

username
[str] The username to use for logging in.

password
[str] The password to use for logging in.

Returns

neo4j._sync.driver.BoltDriver or neo4jGraph, neo4j._sync.driver.Neo4jDriver
The returned neo4j driver.

`static ExportToGraph`(*neo4jGraph*, *cypher=None*, *xMin=-0.5*, *yMin=-0.5*, *zMin=-0.5*, *xMax=0.5*, *yMax=0.5*, *zMax=0.5*, *tolerance=0.0001*, *silent=False*)

Exports the input neo4j graph to a topologic graph.

Parameters

neo4jGraph
[neo4j._sync.driver.BoltDriver or neo4jGraph, neo4j._sync.driver.Neo4jDriver] The input neo4j driver.

cypher
[str, optional] If set to a non-empty string, a Cypher query will be run on the neo4j graph database to return a sub-graph. Default is None.

xMin
[float, optional] The desired minimum value to assign for a vertex's X coordinate. The default is -0.5.

yMin
[float, optional] The desired minimum value to assign for a vertex's Y coordinate. The default is -0.5.

zMin
[float, optional] The desired minimum value to assign for a vertex's Z coordinate. The default is -0.5.

xMax
[float, optional] The desired maximum value to assign for a vertex's X coordinate. The default is 0.5.

yMax
[float, optional] The desired maximum value to assign for a vertex's Y coordinate. The default is 0.5.

zMax
[float, optional] The desired maximum value to assign for a vertex's Z coordinate. The default is 0.5.

silent
[bool, optional] If set to True, no warnings or error messages are displayed. The default is False.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Graph
The output Topologic graph.

`static Reset`(*neo4jGraph*)

Resets the database completely.

Parameters

neo4jGraph
[neo4j._sync.driver.BoltDriver or neo4jGraph, neo4j._sync.driver.Neo4jDriver] The input neo4j driver.

Returns

neo4j._sync.driver.BoltDriver or neo4jGraph, neo4j._sync.driver.Neo4jDriver
The returned neo4j driver.

`static SetGraph`(*neo4jGraph*, *graph*, *labelKey: str = None*, *relationshipKey: str = None*, *bidirectional: bool = True*, *deleteAll: bool = True*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Sets the input topologic graph to the input neo4jGraph.

Parameters

neo4jGraph
[Neo4j.Graph] The input neo4j graph.

graph
[topologic_core.Graph] The input topologic graph.

labelKey
[str , optional] The dictionary key under which to find the vertex's label value. The default is None which means the vertex gets the name 'TopologicGraphVertex'.

relationshipKey
[str , optional] The dictionary key under which to find the edge's relationship value. The default is None which means the edge gets the relationship type 'Connected To'.

bidirectional
[bool , optional] If set to True, the edges in the neo4j graph are set to be bi-drectional.

deleteAll
[bool , optional] If set to True, all previous entities are deleted before adding the new entities.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

neo4jGraph
[TYPE] The input neo4j graph with the input topologic graph added to it.

topologicpy.Plotly module

`class topologicpy.Plotly.Plotly`

Bases: `object`

Methods

AddColorBar(figure[, values, nTicks, ...])	Adds a color bar to the input figure
Colors()	Returns the list of named CSS colors that plotly can use.
DataByDGL(data, labels)	Returns a data frame from the DGL data.
DataByGraph(graph[, sagitta, absolute, ...])	Creates plotly vertex and edge data from the input graph.
DataByTopology(topology[, showVertices, ...])	Creates plotly face, edge, and vertex data.
ExportToImage(figure, path[, format, width, ...])	Exports the plotly figure to an image.
FigureByConfusionMatrix(matrix[, ...])	Returns a Plotly Figure of the input confusion matrix.
FigureByCorrelation(actual, predicted[, ...])	Returns a Plotly Figure showing the correlation between the input actual and predicted values.
FigureByData(data[, width, height, xAxis, ...])	Creates a plotly figure.
FigureByDataFrame(dataFrame[, labels, ...])	Returns a Plotly Figure of the input dataframe
FigureByJSONFile(file)	Imports a plotly figure from a JSON file.
FigureByJSONPath(path)	Imports a plotly figure from a JSON file path.
FigureByMatrix(matrix[, xCategories, ...])	Returns a Plotly Figure of the input matrix.
FigureByPieChart(data, values, names)	Creates a plotly pie chart figure.
FigureByTopology(topology[, showVertices, ...])	Creates a figure from the input topology.
FigureExportToJSON(figure, path[, overwrite])	Exports the input plotly figure to a JSON file.
FigureExportToPDF(figure, path[, width, ...])	Exports the input plotly figure to a PDF file.
FigureExportToPNG(figure, path[, width, ...])	Exports the input plotly figure to a PNG file.
FigureExportToSVG(figure, path[, width, ...])	Exports the input plotly figure to a SVG file.
Renderer()	Return the renderer most suitable for the environment in which the script is running.
Renderers()	Returns a list of the available plotly renderers.
SetCamera(figure[, camera, center, up, ...])	Sets the camera for the input figure.
Show(figure[, camera, center, up, renderer, ...])	Shows the input figure.

ColorScale	
edgeData	
vertexData	

static AddColorBar(*figure*, *values=[]*, *nTicks=5*, *xPosition=-0.15*, *width=15*, *outlineWidth=0*, *title=''*, *subTitle=''*, *units=''*, *colorScale='viridis'*, *mantissa: int = 6*)

Adds a color bar to the input figure

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

values
[list , optional] The input list of values to use for the color bar. The default is [].

nTicks
[int , optional] The number of ticks to use on the color bar. The default is 5.

xPosition
[float , optional] The x location of the color bar. The default is -0.15.

width
[int , optional] The width in pixels of the color bar. The default is 15

outlineWidth
[int , optional] The width in pixels of the outline of the color bar. The default is 0.

title
[str , optional] The title of the color bar. The default is "".

subTitle
[str , optional] The subtitle of the color bar. The default is "".

units: str , optional
The units used in the color bar. The default is ""

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "viridis", "plasma"). The default is "viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/. In addition to these, three color-blind friendly scales are included. These are "protanopia", "deuteranopia", and "tritanopia" for red, green, and blue color-blindness respectively.

mantissa
[int , optional] The desired length of the mantissa for the values listed on the color bar. The default is 6.

Returns
——-
plotly.graph_objs._figure.Figure
The input figure with the color bar added.

`static ColorScale`(*colorScale: str = 'viridis'*)

`static Colors`()

Returns the list of named CSS colors that plotly can use.

Returns

list
The list of named CSS colors.

`static DataByDGL`(*data*, *labels*)

Returns a data frame from the DGL data.

Parameters

data
[list] The data to display.

labels
[list] The labels to use for the data. The data with the labels in this list will be extracted and used in the returned dataFrame.

Returns

pd.DataFrame
A pandas dataFrame

static DataByGraph(*graph*, *sagitta: float = 0*, *absolute: bool = False*, *sides: int = 8*, *angle: float = 0*, *vertexColor: str = 'black'*, *vertexColorKey: str = None*, *vertexSize: float = 6*, *vertexSizeKey: str = None*, *vertexLabelKey: str = None*, *vertexGroupKey: str = None*, *vertexGroups: list = []*, *vertexMinGroup=None*, *vertexMaxGroup=None*, *showVertices: bool = True*, *showVertexLabel: bool = False*, *showVertexLegend: bool = False*, *vertexLegendLabel='Graph Vertices'*, *vertexLegendRank=4*, *vertexLegendGroup=4*, *edgeColor: str = 'black'*, *edgeColorKey: str = None*, *edgeWidth: float = 1*, *edgeWidthKey: str = None*, *edgeLabelKey: str = None*, *edgeGroupKey: str = None*, *edgeGroups: list = []*, *edgeMinGroup=None*, *edgeMaxGroup=None*, *showEdges: bool = True*, *showEdgeLabel: bool = False*, *showEdgeLegend: bool = False*, *edgeLegendLabel='Graph Edges'*, *edgeLegendRank=2*, *edgeLegendGroup=2*, *colorScale: str = 'viridis'*, *mantissa: int = 6*, *silent: bool = False*)

Creates plotly vertex and edge data from the input graph.

Parameters

graph
[topologic_core.Graph] The input graph.

sagitta
[float , optional] The length of the sagitta. In mathematics, the sagitta is the line connecting the center of a chord to the apex (or highest point) of the arc subtended by that chord. The default is 0 which means a straight edge is drawn instead of an arc. The default is 0.

absolute
[bool , optional] If set to True, the sagitta length is treated as an absolute value. Otherwise, it is treated as a ratio based on the length of the edge. The default is False. For example, if the length of the edge is 10, the sagitta is set to 0.5, and absolute is set to False, the sagitta length will be 5. The default is True.

sides
[int , optional] The number of sides of the arc. The default is 8.

vertexColor
[str , optional] The desired color of the output vertices. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

vertexColorKey
[str , optional] The dictionary key under which to find the vertex color. The default is None.

vertexSize
[float , optional] The desired size of the vertices. The default is 6.

vertexLabelKey
[str , optional] The dictionary key to use to display the vertex label. The default is None.

vertexGroupKey
[str , optional] The dictionary key to use to display the vertex group. The default is None.

vertexGroups
[list , optional] The list of vertex groups against which to index the color of the vertex. The default is [].

vertexMinGroup
[int or float , optional] For numeric vertexGroups, vertexMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the

vertexGroupKey. If set to None, it is set to the minimum value in vertexGroups. The default is None.

vertexMaxGroup
[int or float , optional] For numeric vertexGroups, vertexMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the maximum value in vertexGroups. The default is None.

showVertices
[bool , optional] If set to True the vertices will be drawn. Otherwise, they will not be drawn. The default is True.

showVertexLabels
[bool , optional] If set to True, the vertex labels are shown permenantely on screen. Otherwise, they are not. The default is False.

showVertexLegend
[bool , optional] If set to True the vertex legend will be drawn. Otherwise, it will not be drawn. The default is False.

vertexLegendLabel
[str , optional] The legend label string used to identify vertices. The default is "Topology Vertices".

vertexLegendRank
[int , optional] The legend rank order of the vertices of this topology. The default is 1.

vertexLegendGroup
[int , optional] The number of the vertex legend group to which the vertices of this topology belong. The default is 1.

edgeColor
[str , optional] The desired color of the output edges. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

edgeColorKey
[str , optional] The dictionary key under which to find the edge color. The default is None.

edgeWidth
[float , optional] The desired thickness of the output edges. The default is 1.

edgeWidthKey
[str , optional] The dictionary key under which to find the edge width. The default is None.

edgeLabelKey
[str , optional] The dictionary key to use to display the edge label. The default is None.

edgeGroupKey
[str , optional] The dictionary key to use to display the edge group. The default is None.

edgeGroups
[list , optional] The list of groups to use for indexing the color of edges. The default is None.

showEdges
[bool , optional] If set to True the edges will be drawn. Otherwise, they will not be drawn. The default is True.

showEdgeLabels
[bool , optional] If set to True, the edge labels are shown permenantely on screen. Otherwise, they are not. The default is False.

showEdgeLegend
[bool , optional] If set to True the edge legend will be drawn. Otherwise, it will not be drawn. The default is False.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "Viridis", "Plasma"). The default is "Viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

list
The vertex and edge data list.

`static DataByTopology`(*topology*, *showVertices=True*, *vertexSize=1.1*, *vertexSizeKey=None*, *vertexColor='black'*, *vertexColorKey=None*, *vertexLabelKey=None*, *showVertexLabel=False*, *vertexGroupKey=None*, *vertexGroups=[]*, *vertexMinGroup=None*, *vertexMaxGroup=None*, *showVertexLegend=False*, *vertexLegendLabel='Topology Vertices'*, *vertexLegendRank=1*, *vertexLegendGroup=1*, *showEdges=True*, *edgeWidth=1*, *edgeWidthKey=None*, *edgeColor='black'*, *edgeColorKey=None*, *edgeLabelKey=None*, *showEdgeLabel=False*, *edgeGroupKey=None*, *edgeGroups=[]*, *edgeMinGroup=None*, *edgeMaxGroup=None*, *showEdgeLegend=False*, *edgeLegendLabel='Topology Edges'*, *edgeLegendRank=2*, *edgeLegendGroup=2*, *showFaces=True*, *faceOpacity=0.5*, *faceOpacityKey=None*, *faceColor='#FAFAFA'*, *faceColorKey=None*, *faceLabelKey=None*, *faceGroupKey=None*, *faceGroups=[]*, *faceMinGroup=None*, *faceMaxGroup=None*, *showFaceLegend=False*, *faceLegendLabel='Topology Faces'*, *faceLegendRank=3*, *faceLegendGroup=3*, *intensityKey=None*, *intensities=[]*, *colorScale='viridis'*, *mantissa=6*, *tolerance=0.0001*)

Creates plotly face, edge, and vertex data.

Parameters

topology
[topologic_core.Topology] The input topology. This must contain faces and or edges.

showVertices
[bool , optional] If set to True the vertices will be drawn. Otherwise, they will not be drawn. The default is True.

vertexSize
[float , optional] The desired size of the vertices. The default is 1.1.

vertexSizeKey
[str , optional] The dictionary key under which to find the vertex size.The default is None.

vertexColor
[str , optional] The desired color of the output vertices. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

vertexColorKey
[str , optional] The dictionary key under which to find the vertex color.The default is None.

vertexLabelKey
[str , optional] The dictionary key to use to display the vertex label. The default is None.

vertexGroupKey
[str , optional] The dictionary key to use to display the vertex group. The default is None.

vertexGroups
[list , optional] The list of vertex groups against which to index the color of the vertex. The default is [].

vertexMinGroup
[int or float , optional] For numeric vertexGroups, vertexMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the minimum value in vertexGroups. The default is None.

vertexMaxGroup
[int or float , optional] For numeric vertexGroups, vertexMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the maximum value in vertexGroups. The default is None.

showVertexLegend
[bool, optional] If set to True, the legend for the vertices of this topology is shown. Otherwise, it isn't. The default is False.

vertexLegendLabel
[str , optional] The legend label string used to identify vertices. The default is "Topology Vertices".

vertexLegendRank
[int , optional] The legend rank order of the vertices of this topology. The default is 1.

vertexLegendGroup
[int , optional] The number of the vertex legend group to which the vertices of this topology belong. The default is 1.

showEdges
[bool , optional] If set to True the edges will be drawn. Otherwise, they will not be drawn. The default is True.

edgeWidth
[float , optional] The desired thickness of the output edges. The default is 1.

edgeWidthKey
[str , optional] The dictionary key under which to find the edge width.The default is None.

edgeColor
[str , optional] The desired color of the output edges. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

edgeColorKey
[str , optional] The dictionary key under which to find the edge color.The default is None.

edgeLabelKey
[str , optional] The dictionary key to use to display the edge label. The default is None.

edgeGroupKey
[str , optional] The dictionary key to use to display the edge group. The default is None.

edgeGroups
[list , optional] The list of edge groups against which to index the color of the edge. The default is [].

edgeMinGroup
[int or float , optional] For numeric edgeGroups, edgeMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the edgeGroupKey. If set to None, it is set to the minimum value in edgeGroups. The default is None.

edgeMaxGroup
[int or float , optional] For numeric edgeGroups, edgeMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the edgeGroupKey. If set to None, it is set to the maximum value in edgeGroups. The default is None.

showEdgeLegend
[bool, optional] If set to True, the legend for the edges of this topology is shown. Otherwise, it isn't. The default is False.

edgeLegendLabel
[str , optional] The legend label string used to identify edges. The default is "Topology Edges".

edgeLegendRank
[int , optional] The legend rank order of the edges of this topology. The default is 2.

edgeLegendGroup
[int , optional] The number of the edge legend group to which the edges of this topology belong. The default is 2.

showFaces
[bool , optional] If set to True the faces will be drawn. Otherwise, they will not be drawn. The default is True.

faceOpacity
[float , optional] The desired opacity of the output faces (0=transparent, 1=opaque). The default is 0.5.

faceOpacityKey
[str , optional] The dictionary key under which to find the face opacity.The default is None.

faceColor
[str , optional] The desired color of the output faces. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "#FAFAFA".

faceColorKey
[str , optional] The dictionary key under which to find the face color.The default is None.

faceLabelKey
[str , optional] The dictionary key to use to display the face label. The default is None.

faceGroupKey
[str , optional] The dictionary key to use to display the face group. The default is None.

faceGroups
[list , optional] The list of face groups against which to index the color of the face. This can bhave numeric or string values. This should match the type of value associated with the faceGroupKey. The default is [].

faceMinGroup
[int or float , optional] For numeric faceGroups, minGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the faceGroupKey. If set to None, it is set to the minimum value in faceGroups. The default is None.

faceMaxGroup
[int or float , optional] For numeric faceGroups, maxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the faceGroupKey. If set to None, it is set to the maximum value in faceGroups. The default is None.

showFaceLegend
[bool, optional] If set to True, the legend for the faces of this topology is shown. Otherwise, it isn't. The default is False.

faceLegendLabel
[str , optional] The legend label string used to idenitfy edges. The default is "Topology Faces".

faceLegendRank
[int , optional] The legend rank order of the faces of this topology. The default is 3.

faceLegendGroup
[int , optional] The number of the face legend group to which the faces of this topology belong. The default is 3.

intensityKey: str, optional
If not None, the dictionary of each vertex is searched for the value associated with the intensity key. This value is then used to color-code the vertex based on the colorScale. The default is None.

intensities
[list , optional] The list of intensities against which to index the intensity of the vertex. The default is [].

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "Viridis", "Plasma"). The default is "Viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The vertex, edge, and face data list.

`static ExportToImage`(*figure*, *path*, *format='png'*, *width='1920'*, *height='1080'*)

Exports the plotly figure to an image.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

path
[str] The image file path.

format
[str , optional] The desired format. This can be any of "jpg", "jpeg", "pdf", "png", "svg", or "webp". It is case insensitive. The default is "png".

width
[int , optional] The width in pixels of the figure. The default value is 1920.

height
[int , optional] The height in pixels of the figure. The default value is 1080.

Returns

bool
True if the image was exported sucessfully. False otherwise.

`static FigureByConfusionMatrix`(*matrix*, *categories=[]*, *minValue=None*, *maxValue=None*, *title='Confusion Matrix'*, *xTitle='Actual Categories'*, *yTitle='Predicted Categories'*, *width=950*, *height=500*, *showScale=True*, *colorScale='viridis'*, *colorSamples=10*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=40*, *marginBottom=0*)

Returns a Plotly Figure of the input confusion matrix. Actual categories are displayed on the X-Axis, Predicted categories are displayed on the Y-Axis.

Parameters

matrix
[list or numpy.array] The matrix to display.

categories
[list] The list of categories to use on the X and Y axes.

minValue
[float , optional] The desired minimum value to use for the color scale. If set to None, the minmum value found in the input matrix will be used. The default is None.

maxValue
[float , optional] The desired maximum value to use for the color scale. If set to None, the maximum value found in the input matrix will be used. The default is None.

title
[str , optional] The desired title to display. The default is "Confusion Matrix".

xTitle
[str , optional] The desired X-axis title to display. The default is "Actual Categories".

yTitle
[str , optional] The desired Y-axis title to display. The default is "Predicted Categories".

width
[int , optional] The desired width of the figure. The default is 950.

height
[int , optional] The desired height of the figure. The default is 500.

showScale
[bool , optional] If set to True, a color scale is shown on the right side of the figure. The default is True.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "Viridis", "Plasma"). The default is "Viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

colorSamples
[int , optional] The number of discrete color samples to use for displaying the data. The default is 10.

backgroundColor
[list or str , optional] The desired background color. This can be any color list or plotly color string and may be specified as: - An rgb list (e.g. [255,0,0]) - A cmyk list (e.g. [0.5, 0, 0.25, 0.2]) - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

marginLeft
[int , optional] The desired left margin in pixels. The default is 0.

marginRight
[int , optional] The desired right margin in pixels. The default is 0.

marginTop
[int , optional] The desired top margin in pixels. The default is 40.

marginBottom
[int , optional] The desired bottom margin in pixels. The default is 0.

Returns

plotly.Figure
The created plotly figure.

`static FigureByCorrelation`(*actual*, *predicted*, *title='Correlation between Actual and Predicted Values'*, *xTitle='Actual Values'*, *yTitle='Predicted Values'*, *dotColor='blue'*, *lineColor='red'*, *width=800*, *height=600*, *theme='default'*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=40*, *marginBottom=0*)

Returns a Plotly Figure showing the correlation between the input actual and predicted values. Actual values are displayed on the X-Axis, Predicted values are displayed on the Y-Axis.

Parameters

actual
[list] The actual values to display.

predicted
[list] The predicted values to display.

title
[str , optional] The desired title to display. The default is "Correlation between Actual and Predicted Values".

xTitle
[str , optional] The desired X-axis title to display. The default is "Actual Values".

yTitle
[str , optional] The desired Y-axis title to display. The default is "Predicted Values".

dotColor
[str , optional] The desired color of the dots. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'blue'.

lineColor
[str , optional] The desired color of the best fit line. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'red'.

width
[int , optional] The desired width of the figure. The default is 800.

height
[int , optional] The desired height of the figure. The default is 600.

theme
[str , optional] The plotly color scheme to use. The options are "dark", "light", "default". The default is "default".

backgroundColor
[list or str , optional] The desired background color. This can be any color list or plotly color string and may be specified as: - An rgb list (e.g. [255,0,0]) - A cmyk list (e.g. [0.5, 0, 0.25, 0.2]) - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

marginLeft
[int , optional] The desired left margin in pixels. The default is 0.

marginRight
[int , optional] The desired right margin in pixels. The default is 0.

marginTop
[int , optional] The desired top margin in pixels. The default is 40.

marginBottom
[int , optional] The desired bottom margin in pixels. The default is 0.

Returns

plotly.Figure
The created plotly figure.

`static FigureByData`(*data*, *width=950*, *height=500*, *xAxis=False*, *yAxis=False*, *zAxis=False*, *axisSize=1*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=20*, *marginBottom=0*, *tolerance=0.0001*)

Creates a plotly figure.

Parameters

data
[list] The input list of plotly data.

width
[int , optional] The width in pixels of the figure. The default value is 950.

height
[int , optional] The height in pixels of the figure. The default value is 950.

xAxis
[bool , optional] If set to True the x axis is drawn. Otherwise it is not drawn. The default is False.

yAxis
[bool , optional] If set to True the y axis is drawn. Otherwise it is not drawn. The default is False.

zAxis
[bool , optional] If set to True the z axis is drawn. Otherwise it is not drawn. The default is False.

axisSize
[float , optional] The size of the X, Y, Z, axes. The default is 1.

backgroundColor
[list or str , optional] The desired background color. This can be any color list or plotly color string and may be specified as: - An rgb list (e.g. [255,0,0]) - A cmyk list (e.g. [0.5, 0, 0.25, 0.2]) - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

marginLeft
[int , optional] The size in pixels of the left margin. The default value is 0.

marginRight
[int , optional] The size in pixels of the right margin. The default value is 0.

marginTop
[int , optional] The size in pixels of the top margin. The default value is 20.

marginBottom
[int , optional] The size in pixels of the bottom margin. The default value is 0.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

plotly.graph_objs._figure.Figure
The created plotly figure.

`static FigureByDataFrame`(*dataFrame*, *labels=[]*, *width=950*, *height=500*, *title='Untitled'*, *xTitle='X Axis'*, *xSpacing=1*, *yTitle='Y Axis'*, *ySpacing=1.0*, *useMarkers=False*, *chartType='Line'*, *backgroundColor='rgba(0,0,0,0)'*, *gridColor='lightgray'*, *marginLeft=0*, *marginRight=0*, *marginTop=40*, *marginBottom=0*)

Returns a Plotly Figure of the input dataframe

Parameters

df
[pandas.df] The pandas dataframe to display.

data_labels
[list] The labels to use for the data.

width
[int , optional] The desired width of the figure. The default is 950.

height
[int , optional] The desired height of the figure. The default is 500.

title
[str , optional] The chart title. The default is "Training and Testing Results".

xTitle
[str , optional] The X-axis title. The default is "Epochs".

xSpacing
[float , optional] The X-axis spacing. The default is 1.0.

yTitle
[str , optional] The Y-axis title. The default is "Accuracy and Loss".

ySpacing
[float , optional] The Y-axis spacing. The default is 0.1.

useMarkers
[bool , optional] If set to True, markers will be displayed. The default is False.

chartType
[str , optional] The desired type of chart. The options are "Line", "Bar", or "Scatter". It is case insensitive. The default is "Line".

backgroundColor
[list or str , optional] The desired background color. This can be any color list or plotly color string and may be specified as: - An rgb list (e.g. [255,0,0]) - A cmyk list (e.g. [0.5, 0, 0.25, 0.2]) - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

grid
[str , optional] The desired background color. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'lightgray'

marginLeft
[int , optional] The desired left margin in pixels. The default is 0.

marginRight
[int , optional] The desired right margin in pixels. The default is 0.

marginTop
[int , optional] The desired top margin in pixels. The default is 40.

marginBottom
[int , optional] The desired bottom margin in pixels. The default is 0.

Returns

None.

`static FigureByJSONFile`(*file*)

Imports a plotly figure from a JSON file.

Parameters

file
[file object] The JSON file.

Returns

plotly.graph_objs._figure.Figure
The imported figure.

static FigureByJSONPath(*path*)

Imports a plotly figure from a JSON file path.

Parameters

path
[str] The path to the BRep file.

Returns

plotly.graph_objs._figure.Figure
The imported figure.

static FigureByMatrix(*matrix*, *xCategories=[]*, *yCategories=[]*, *minValue=None*, *maxValue=None*, *title='Matrix'*, *xTitle='X Axis'*, *yTitle='Y Axis'*, *width=950*, *height=950*, *showScale=False*, *colorScale='gray'*, *colorSamples=10*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=40*, *marginBottom=0*, *mantissa: int = 6*)

Returns a Plotly Figure of the input matrix.

Parameters

matrix
[list or numpy.array] The matrix to display.

categories
[list] The list of categories to use on the X and Y axes.

minValue
[float , optional] The desired minimum value to use for the color scale. If set to None, the minmum value found in the input matrix will be used. The default is None.

maxValue
[float , optional] The desired maximum value to use for the color scale. If set to None, the maximum value found in the input matrix will be used. The default is None.

title
[str , optional] The desired title to display. The default is "Confusion Matrix".

xTitle
[str , optional] The desired X-axis title to display. The default is "Actual".

yTitle
[str , optional] The desired Y-axis title to display. The default is "Predicted".

width
[int , optional] The desired width of the figure. The default is 950.

height
[int , optional] The desired height of the figure. The default is 500.

showScale
[bool , optional] If set to True, a color scale is shown on the right side of the figure. The default is True.

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "Viridis", "Plasma"). The default is "Viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

colorSamples
[int , optional] The number of discrete color samples to use for displaying the data. The default is 10.

backgroundColor
[list or str , optional] The desired background color. This can be any color list or plotly color string and may be specified as: - An rgb list (e.g. [255,0,0]) - A cmyk list (e.g. [0.5, 0, 0.25, 0.2]) - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

marginLeft
[int , optional] The desired left margin in pixels. The default is 0.

marginRight
[int , optional] The desired right margin in pixels. The default is 0.

marginTop
[int , optional] The desired top margin in pixels. The default is 40.

marginBottom
[int , optional] The desired bottom margin in pixels. The default is 0.

mantissa
[int , optional] The desired number of digits of the mantissa. The default is 6.

`static FigureByPieChart`(*data*, *values*, *names*)

Creates a plotly pie chart figure.

Parameters

data
[list] The input list of plotly data.

values
[list] The input list of values.

names
[list] The input list of names.

`static FigureByTopology`(*topology*, *showVertices=True*, *vertexSize=1.1*, *vertexColor='black'*, *vertexLabelKey=None*, *vertexGroupKey=None*, *vertexGroups=[]*, *vertexMinGroup=None*, *vertexMaxGroup=None*, *showVertexLegend=False*, *vertexLegendLabel='Topology Vertices'*, *vertexLegendRank=1*, *vertexLegendGroup=1*, *showEdges=True*, *edgeWidth=1*, *edgeColor='black'*, *edgeLabelKey=None*, *edgeGroupKey=None*, *edgeGroups=[]*, *edgeMinGroup=None*, *edgeMaxGroup=None*, *showEdgeLegend=False*, *edgeLegendLabel='Topology Edges'*, *edgeLegendRank=2*, *edgeLegendGroup=2*, *showFaces=True*, *faceOpacity=0.5*, *faceColor='#FAFAFA'*, *faceLabelKey=None*, *faceGroupKey=None*, *faceGroups=[]*, *faceMinGroup=None*, *faceMaxGroup=None*, *showFaceLegend=False*, *faceLegendLabel='Topology Faces'*, *faceLegendRank=3*, *faceLegendGroup=3*, *intensityKey=None*, *width=950*, *height=500*, *xAxis=False*, *yAxis=False*, *zAxis=False*, *axisSize=1*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=20*, *marginBottom=0*, *showScale=False*, *cbValues=[]*, *cbTicks=5*, *cbX=-0.15*, *cbWidth=15*, *cbOutlineWidth=0*, *cbTitle=''*, *cbSubTitle=''*, *cbUnits=''*, *colorScale='viridis'*, *mantissa=6*, *tolerance=0.0001*)

Creates a figure from the input topology.

Parameters

topology
[topologic_core.Topology] The input topology. This must contain faces and or edges.

showVertices
[bool , optional] If set to True the vertices will be drawn. Otherwise, they will not be drawn. The default is True.

vertexSize
[float , optional] The desired size of the vertices. The default is 1.1.

vertexColor
[str , optional] The desired color of the output vertices. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

vertexLabelKey
[str , optional] The dictionary key to use to display the vertex label. The default is None.

vertexGroupKey
[str , optional] The dictionary key to use to display the vertex group. The default is None.

vertexGroups
[list , optional] The list of vertex groups against which to index the color of the vertex. The default is [].

vertexMinGroup
[int or float , optional] For numeric vertexGroups, vertexMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the minimum value in vertexGroups. The default is None.

edgeMaxGroup
[int or float , optional] For numeric vertexGroups, vertexMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the maximum value in vertexGroups. The default is None.

showVertexLegend
[bool, optional] If set to True, the legend for the vertices of this topology is shown. Otherwise, it isn't. The default is False.

vertexLegendLabel
[str , optional] The legend label string used to identify vertices. The default is "Topology Vertices".

vertexLegendRank
[int , optional] The legend rank order of the vertices of this topology. The default is 1.

vertexLegendGroup
[int , optional] The number of the vertex legend group to which the vertices of this topology belong. The default is 1.

showEdges
[bool , optional] If set to True the edges will be drawn. Otherwise, they will not be drawn. The default is True.

edgeWidth
[float , optional] The desired thickness of the output edges. The default is 1.

edgeColor
[str , optional] The desired color of the output edges. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

edgeLabelKey
[str , optional] The dictionary key to use to display the edge label. The default is None.

edgeGroupKey
[str , optional] The dictionary key to use to display the edge group. The default is None.

edgeGroups
[list , optional] The list of edge groups against which to index the color of the edge. The default is [].

edgeMinGroup
[int or float , optional] For numeric edgeGroups, edgeMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the edgeGroupKey. If set to None, it is set to the minimum value in edgeGroups. The default is None.

edgeMaxGroup
[int or float , optional] For numeric edgeGroups, edgeMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the edgeGroupKey. If set to None, it is set to the maximum value in edgeGroups. The default is None.

showEdgeLegend
[bool, optional] If set to True, the legend for the edges of this topology is shown. Otherwise, it isn't. The default is False.

edgeLegendLabel
[str , optional] The legend label string used to identify edges. The default is "Topology Edges".

edgeLegendRank
[int , optional] The legend rank order of the edges of this topology. The default is 2.

edgeLegendGroup
[int , optional] The number of the edge legend group to which the edges of this topology belong. The default is 2.

showFaces
[bool , optional] If set to True the faces will be drawn. Otherwise, they will not be drawn. The default is True.

faceOpacity
[float , optional] The desired opacity of the output faces (0=transparent, 1=opaque). The default is 0.5.

faceColor
[str , optional] The desired color of the output faces. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "#FAFAFA".

faceLabelKey
[str , optional] The dictionary key to use to display the face label. The default is None.

faceGroupKey
[str , optional] The dictionary key to use to display the face group. The default is None.

faceGroups
[list , optional] The list of face groups against which to index the color of the face. This can bhave numeric or string values. This should match the type of value associated with the faceGroupKey. The default is [].

faceMinGroup
[int or float , optional] For numeric faceGroups, minGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the faceGroupKey. If set to None, it is set to the minimum value in faceGroups. The default is None.

faceMaxGroup
[int or float , optional] For numeric faceGroups, maxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the faceGroupKey. If set to None, it is set to the maximum value in faceGroups. The default is None.

showFaceLegend
[bool, optional] If set to True, the legend for the faces of this topology is shown. Otherwise, it isn't. The default is False.

faceLegendLabel
[str , optional] The legend label string used to idenitfy edges. The default is "Topology Faces".

faceLegendRank
[int , optional] The legend rank order of the faces of this topology. The default is 3.

faceLegendGroup
[int , optional] The number of the face legend group to which the faces of this topology belong. The default is 3.

width
[int , optional] The width in pixels of the figure. The default value is 950.

height
[int , optional] The height in pixels of the figure. The default value is 950.

xAxis
[bool , optional] If set to True the x axis is drawn. Otherwise it is not drawn. The default is False.

yAxis
[bool , optional] If set to True the y axis is drawn. Otherwise it is not drawn. The default is False.

zAxis
[bool , optional] If set to True the z axis is drawn. Otherwise it is not drawn. The default is False.

backgroundColor
[list or str , optional] The desired background color. This can be any color list or plotly color string and may be specified as: - An rgb list (e.g. [255,0,0]) - A cmyk list (e.g. [0.5, 0, 0.25, 0.2]) - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

marginLeft
[int , optional] The size in pixels of the left margin. The default value is 0.

marginRight
[int , optional] The size in pixels of the right margin. The default value is 0.

marginTop
[int , optional] The size in pixels of the top margin. The default value is 20.

marginBottom
[int , optional] The size in pixels of the bottom margin. The default value is 0.

camera
[list , optional] The desired location of the camera). The default is [-1.25, -1.25, 1.25].

center
[list , optional] The desired center (camera target). The default is [0, 0, 0].

up
[list , optional] The desired up vector. The default is [0, 0, 1].

renderer
[str , optional] The desired renderer. See Plotly.Renderers(). The default is "notebook".

intensityKey
[str , optional] If not None, the dictionary of each vertex is searched for the value associated with the intensity key. This value is then used to color-code the vertex based on the colorScale. The default is None.

showScale
[bool , optional] If set to True, the colorbar is shown. The default is False.

cbValues
[list , optional] The input list of values to use for the colorbar. The default is [].

cbTicks
[int , optional] The number of ticks to use on the colorbar. The default is 5.

cbX
[float , optional] The x location of the colorbar. The default is -0.15.

cbWidth
[int , optional] The width in pixels of the colorbar. The default is 15

cbOutlineWidth
[int , optional] The width in pixels of the outline of the colorbar. The default is 0.

cbTitle
[str , optional] The title of the colorbar. The default is "".

cbSubTitle
[str , optional] The subtitle of the colorbar. The default is "".

cbUnits: str , optional
The units used in the colorbar. The default is ""

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "viridis", "plasma"). The default is "viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

mantissa
[int , optional] The desired length of the mantissa for the values listed on the colorbar. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

Plotly figure

`static FigureExportToJSON`(*figure*, *path*, *overwrite=False*)

Exports the input plotly figure to a JSON file.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

path
[str] The input file path.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't.

Returns

bool
True if the export operation is successful. False otherwise.

`static FigureExportToPDF`(*figure*, *path*, *width=1920*, *height=1200*, *overwrite=False*)

Exports the input plotly figure to a PDF file.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

path
[str] The input file path.

width
[int, optional] The width of the exported image in pixels. The default is 1920.

height
[int , optional] The height of the exported image in pixels. The default is 1200.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't.

Returns

bool
True if the export operation is successful. False otherwise.

`static FigureExportToPNG`(*figure*, *path*, *width=1920*, *height=1200*, *overwrite=False*)

Exports the input plotly figure to a PNG file.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

path
[str] The input file path.

width
[int, optional] The width of the exported image in pixels. The default is 1920.

height
[int , optional] The height of the exported image in pixels. The default is 1200.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't.

Returns

bool
True if the export operation is successful. False otherwise.

`static FigureExportToSVG`(*figure*, *path*, *width=1920*, *height=1200*, *overwrite=False*)

Exports the input plotly figure to a SVG file.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

path
[str] The input file path.

width
[int, optional] The width of the exported image in pixels. The default is 1920.

height
[int , optional] The height of the exported image in pixels. The default is 1200.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't.

Returns

bool
True if the export operation is successful. False otherwise.

`static Renderer`()

Return the renderer most suitable for the environment in which the script is running.

Parameters

Returns

str
The most suitable renderer type for the environment in which the script is running. Currently, this is limited to: - "vscode" if running in Visual Studio Code - "colab" if running in Google Colab - "iframe" if running in jupyter notebook or jupyterlab - "browser" if running in anything else

`static Renderers`()

Returns a list of the available plotly renderers.

Parameters

Returns

list
The list of the available plotly renderers.

`static SetCamera`(*figure*, *camera=[-1.25, -1.25, 1.25]*, *center=[0, 0, 0]*, *up=[0, 0, 1]*, *projection='perspective'*)

Sets the camera for the input figure.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

camera
[list , optional] The desired location of the camera. The default is [-1.25, -1.25, 1.25].

center
[list , optional] The desired center (camera target). The default is [0, 0, 0].

up
[list , optional] The desired up vector. The default is [0, 0, 1].

projection
[str , optional] The desired type of projection. The options are "orthographic" or "perspective". It is case insensitive. The default is "perspective"

Returns

plotly.graph_objs._figure.Figure
The updated figure

`static Show`(*figure*, *camera=[-1.25, -1.25, 1.25]*, *center=[0, 0, 0]*, *up=[0, 0, 1]*, *renderer=None*, *projection='perspective'*)

Shows the input figure.

Parameters

figure
[plotly.graph_objs._figure.Figure] The input plotly figure.

camera
[list , optional] The desired location of the camera. The default is [0, 0, 0].

center
[list , optional] The desired center (camera target). The default is [0, 0, 0].

up
[list , optional] The desired up vector. The default is [0, 0, 1].

renderer
[str , optional] The desired renderer. See Plotly.Renderers(). If set to None, the code will attempt to discover the most suitable renderer. The default is None.

projection
[str, optional] The desired type of projection. The options are "orthographic" or "perspective". It is case insensitive. The default is "perspective"

Returns

None

static edgeData(*vertices*, *edges*, *dictionaries=None*, *color='black'*, *colorKey=None*, *width=1*, *widthKey=None*, *labelKey=None*, *showEdgeLabel=False*, *groupKey=None*, *minGroup=None*, *maxGroup=None*, *groups=[]*, *legendLabel='Topology Edges'*, *legendGroup=2*, *legendRank=2*, *showLegend=True*, *colorScale='Viridis'*)

static vertexData(*vertices*, *dictionaries=[]*, *color='black'*, *colorKey=None*, *size=1.1*, *sizeKey=None*, *labelKey=None*, *showVertexLabel=False*, *groupKey=None*, *minGroup=None*, *maxGroup=None*, *groups=[]*, *legendLabel='Topology Vertices'*, *legendGroup=1*, *legendRank=1*, *showLegend=True*, *colorScale='Viridis'*)

topologicpy.Polyskel module

class topologicpy.Polyskel.**Debug**(*image*)

Bases: object

Methods

line	
rectangle	
show	

line(**args*, ***kwargs*)

rectangle(**args*, ***kwargs*)

show()

class topologicpy.Polyskel.**Line2**(*p1*, *p2=None*)

Bases: object

Methods

distance	
intersect	

distance(*point*)

intersect(*other*)

class topologicpy.Polyskel.**LineSegment2**(*p1*, *p2*)

Bases: *Line2*

Methods

distance	
intersect	

intersect(*other*)

class topologicpy.Polyskel.**Point2**(*x*, *y*)

Bases: object

Methods

cross(other)	Computes the cross product of this point with another point.

distance	
dot	
normalized	

cross(*other*)

Computes the cross product of this point with another point.

Args:
other (Point2): The other point to compute the cross product with.

Returns:
float: The cross product value.

distance(*other*)

dot(*other*)

normalized()

class topologicpy.Polyskel.**Ray2**(*p*, *v*)

Bases: object

Methods

intersect(other)	Intersects this ray with another ray.

intersect(*other*)

Intersects this ray with another ray.

Args:
other (Ray2): The other ray to intersect with.

Returns:
Point2 or None: The intersection point if it exists, or None if the rays do not intersect.

class topologicpy.Polyskel.**Subtree**(*source*, *height*, *sinks*)

Bases: tuple

Attributes

height
Alias for field number 1

sinks
Alias for field number 2

source
Alias for field number 0

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

height

Alias for field number 1

sinks

Alias for field number 2

source

Alias for field number 0

topologicpy.Polyskel.**set_debug**(*image*)

topologicpy.Polyskel.**skeletonize**(*polygon*, *holes=None*)

Compute the straight skeleton of a polygon.

The polygon should be given as a list of vertices in counter-clockwise order. Holes is a list of the contours of the holes, the vertices of which should be in clockwise order.

Please note that the y-axis goes downwards as far as polyskel is concerned, so specify your ordering accordingly.

Returns the straight skeleton as a list of "subtrees", which are in the form of (source, height, sinks), where source is the highest points, height is its height, and sinks are the point connected to the source.

topologicpy.PyG module

class topologicpy.PyG.**CustomGraphDataset**(*root=None*, *data_list=None*, *indices=None*, *node_level=False*, *graph_level=True*, *node_attr_key='feat'*, *edge_attr_key='feat'*)

Bases: Dataset

Attributes

has_download
Checks whether the dataset defines a download() method.

has_process
Checks whether the dataset defines a process() method.

num_classes
Returns the number of classes in the dataset.

num_edge_features
Returns the number of features per edge in the dataset.

num_features
Returns the number of features per node in the dataset.

num_node_features
Returns the number of features per node in the dataset.

processed_dir
processed_file_names
The name of the files in the `self.processed_dir` folder that must be present in order to skip processing.

processed_paths
The absolute filepaths that must be present in order to skip processing.

raw_dir
raw_file_names
The name of the files in the `self.raw_dir` folder that must be present in order to skip downloading.

raw_paths
The absolute filepaths that must be present in order to skip downloading.

Methods

download()	Downloads the dataset to the `self.raw_dir` folder.
get(idx)	Gets the data object at index `idx`.
get_summary()	Collects summary statistics for the dataset.
index_select(idx)	Creates a subset of the dataset from specified indices `idx`.
len()	Returns the number of data objects stored in the dataset.
print_summary([fmt])	Prints summary statistics of the dataset to the console.
process()	Processes the dataset to the `self.processed_dir` folder.
shuffle([return_perm])	Randomly shuffles the examples in the dataset.
to_datapipe()	Converts the dataset into a `torch.utils.data.DataPipe`.

indices	
process_all	

process_all()

class topologicpy.PyG.**PyG**
Bases: object

Methods

Accuracy(actual, predicted[, mantissa])	Computes the accuracy of the input predictions based on the input labels.
ConfusionMatrix(actual, predicted[, normalize])	Returns the confusion matrix for the input actual and predicted labels.
DatasetByCSVPath(path[, ...])	Returns PyTorch Geometric dataset according to the input CSV folder path.
DatasetGraphLabels(dataset[, graphLabelHeader])	Returns the labels of the graphs in the input dataset
DatasetSplit(dataset[, split, shuffle, ...])	Splits the dataset into three subsets.
Hyperparameters(optimizer[, model_type, ...])	Creates a hyperparameters object based on the input settings.
MSE(actual, predicted[, mantissa])	Computes the Mean Squared Error (MSE) of the input predictions based on the input labels.
Model(hparams, trainingDataset[, ...])	Creates a neural network classifier.
ModelClassify(model, dataset[, nodeATTRKey])	Predicts the classification the labels of the input dataset.
ModelData(model)	Returns the data of the model
ModelLoad(path, model)	Returns the model found at the input file path.
ModelPredict(model, dataset[, nodeATTRKey])	Predicts the value of the input dataset.
ModelSave(model, path[, overwrite])	Saves the model.
ModelTest(model)	Tests the neural network model.
ModelTrain(model)	Trains the neural network model.
Optimizer([name, amsgrad, betas, eps, lr, ...])	Returns the parameters of the optimizer
Performance(actual, predicted[, mantissa])	Computes regression model performance measures.
Show(data, labels[, title, xTitle, ...])	Shows the data in a plolty graph.

static Accuracy(*actual*, *predicted*, *mantissa: int = 6*)

Computes the accuracy of the input predictions based on the input labels. This is to be used only with classification not with regression.

Parameters

actual
[list] The input list of actual values.

predicted
[list] The input list of predicted values.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
A dictionary returning the accuracy information. This contains the following keys and values: - “accuracy” (float): The number of correct predictions divided by the length of the list. - “correct” (int): The number of correct predictions - “mask” (list): A boolean mask for correct vs. wrong predictions which can be used to filter the list of predictions - “size” (int): The size of the predictions list - “wrong” (int): The number of wrong predictions

static ConfusionMatrix(*actual*, *predicted*, *normalize=False*)

Returns the confusion matrix for the input actual and predicted labels. This is to be used with classification tasks only not regression.

Parameters

actual
[list] The input list of actual labels.

predicted
[list] The input list of predicts labels.

normalized
[bool , optional] If set to True, the returned data will be normalized (proportion of 1). Otherwise, actual numbers are returned. The default is False.

Returns

list
The created confusion matrix.

`static DatasetByCSVPath`(*path*, *numberOfGraphClasses=0*, *nodeATTRKey='feat'*, *edgeATTRKey='feat'*, *nodeOneHotEncode=False*, *nodeFeaturesCategories=[]*, *edgeOneHotEncode=False*, *edgeFeaturesCategories=[]*, *addSelfLoop=False*, *node_level=False*, *graph_level=True*)

Returns PyTorch Geometric dataset according to the input CSV folder path. The folder must contain "graphs.csv", "edges.csv", "nodes.csv", and "meta.yml" files according to conventions.

Parameters

path
[str] The path to the folder containing the necessary CSV and YML files.

Returns

PyG Dataset
The PyG dataset

`static DatasetGraphLabels`(*dataset*, *graphLabelHeader='label'*)

Returns the labels of the graphs in the input dataset

Parameters

dataset
[CustomDataset] The input dataset

graphLabelHeader: str , optional
The key string under which the graph labels are stored. The default is "label".

Returns

list
The list of graph labels.

`static DatasetSplit`(*dataset*, *split=[0.8, 0.1, 0.1]*, *shuffle=True*, *randomState=42*)

Splits the dataset into three subsets.

Parameters

dataset
[CustomDataset] The input dataset

split: list , optional
The list of ratios. This list must be made out of three numbers adding to 1.

shuffle: boolean , optional
If set to True, the subsets are created from random indices. Otherwise, they are split sequentially. The default is True.

randomState
[int , optional] The random seed to use for reproducibility. The default is 42.

Returns

list
The list of three subset datasets.

`static Hyperparameters`(*optimizer*, *model_type='classifier'*, *cv_type='Holdout'*, *split=[0.8, 0.1, 0.1]*, *k_folds=5*, *hl_widths=[32]*, *conv_layer_type='SAGEConv'*, *pooling='AvgPooling'*, *batch_size=1*, *epochs=1*, *use_gpu=False*, *loss_function='Cross Entropy'*, *input_type='graph'*)

Creates a hyperparameters object based on the input settings.

Parameters

model_type
[str , optional] The desired type of model. The options are: - "Classifier" - "Regressor" The option is case insensitive. The default is "classifierholdout"

optimizer
[Optimizer] The desired optimizer.

cv_type
[str , optional] The desired cross-validation method. This can be "Holdout" or "K-Fold". It is case-insensitive. The default is "Holdout".

split
[list , optional] The desired split between training validation, and testing. [0.8, 0.1, 0.1] means that 80% of the data is used for training 10% of the data is used for validation, and 10% is used for testing. The default is [0.8, 0.1, 0.1].

k_folds
[int , optional] The desired number of k-folds. The default is 5.

hl_widths
[list , optional] The list of hidden layer widths. A list of [16, 32, 16] means that the model will have 3 hidden layers with number of neurons in each being 16, 32, 16 respectively from input to output. The default is [32].

conv_layer_type
[str , optional] The desired type of the convolution layer. The options are "Classic", "GraphConv", "GINConv", "SAGEConv", "TAGConv", "DGN". It is case insensitive. The default is "SAGEConv".

pooling
[str , optional] The desired type of pooling. The options are "AvgPooling", "MaxPooling", or "SumPooling". It is case insensitive. The default is "AvgPooling".

batch_size
[int , optional] The desired batch size. The default is 1.

epochs
[int , optional] The desired number of epochs. The default is 1.

use_gpu
[bool , optional] If set to True, the model will attempt to use the GPU. The default is False.

loss_function
[str , optional] The desired loss function. The options are "Cross-Entropy" or "Negative Log Likelihood". It is case insensitive. The default is "Cross-Entropy".

input_type
[str] selects the input_type of model such as graph, node or edge

Returns
——-

Hyperparameters
The created hyperparameters object.

`static MSE`(*actual*, *predicted*, *mantissa: int = 6*)

Computes the Mean Squared Error (MSE) of the input predictions based on the input labels. This is to be used with regression models.

Parameters

actual
[list] The input list of actual values.

predicted
[list] The input list of predicted values.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
A dictionary returning the MSE information. This contains the following keys and values: - “mse” (float): The mean squared error rounded to the specified mantissa. - “size” (int): The size of the predictions list.

`static Model`(*hparams*, *trainingDataset*, *validationDataset=None*, *testingDataset=None*)

Creates a neural network classifier.

Parameters

hparams
[HParams] The input hyperparameters

trainingDataset
[CustomDataset] The input training dataset.

validationDataset
[CustomDataset] The input validation dataset. If not specified, a portion of the trainingDataset will be used for validation according the to the split list as specified in the hyperparameters.

testingDataset
[CustomDataset] The input testing dataset. If not specified, a portion of the trainingDataset will be used for testing according the to the split list as specified in the hyper-parameters.

Returns

Classifier
The created classifier

`static ModelClassify`(*model*, *dataset*, *nodeATTRKey='feat'*)

Predicts the classification the labels of the input dataset.

Parameters

dataset
[PyGDataset] The input PyG dataset.

model
[Model] The input trained model.

nodeATTRKey
[str , optional] The key used for node attributes. The default is "feat".

Returns

dict
Dictionary containing labels and probabilities. The included keys and values are: - "predictions" (list): the list of predicted labels - "probabilities" (list): the list of probabilities that the label is one of the categories.

`static ModelData`(*model*)

Returns the data of the model

Parameters

model
[Model] The input model.

Returns

dict
A dictionary containing the model data. The keys in the dictionary are: 'Model Type' 'Optimizer' 'CV Type' 'Split' 'K-Folds' 'HL Widths' 'Conv Layer Type' 'Pooling' 'Learning Rate' 'Batch Size' 'Epochs' 'Training Accuracy' 'Validation Accuracy' 'Testing Accuracy' 'Training Loss' 'Validation Loss' 'Testing Loss' 'Accuracies' (Classifier and K-Fold only) 'Max Accuracy' (Classifier and K-Fold only) 'Losses' (Regressor and K-fold only) 'min Loss' (Regressor and K-fold only)

`static ModelLoad`(*path*, *model*)

Returns the model found at the input file path.

Parameters

path
[str] File path for the saved classifier.

model
[torch.nn.module] Initialized instance of model

Returns

PyG Classifier
The classifier.

`static ModelPredict`(*model*, *dataset*, *nodeATTRKey='feat'*)

Predicts the value of the input dataset.

Parameters

dataset
[PyGDataset] The input PyG dataset.

model
[Model] The input trained model.

nodeATTRKey
[str , optional] The key used for node attributes. The default is "feat".

Returns

list
The list of predictions

`static ModelSave`(*model*, *path*, *overwrite=False*)

Saves the model.

Parameters

model
[Model] The input model.

path
[str] The file path at which to save the model.

overwrite
[bool, optional] If set to True, any existing file will be overwritten. Otherwise, it won't. The default is False.

Returns

bool
True if the model is saved correctly. False otherwise.

`static ModelTest`(*model*)

Tests the neural network model.

Parameters

model
[Model] The input model.

Returns

Model
The tested model

`static ModelTrain`(*model*)

Trains the neural network model.

Parameters

model
[Model] The input model.

Returns

Model
The trained model

`static Optimizer`(*name='Adam'*, *amsgrad=True*, *betas=(0.9, 0.999)*, *eps=1e-06*, *lr=0.001*, *maximize=False*, *weightDecay=0.0*, *rho=0.9*, *lr_decay=0.0*)

Returns the parameters of the optimizer

Parameters

amsgrad
[bool , optional.] amsgrad is an extension to the Adam version of gradient descent that attempts to improve the convergence properties of the algorithm, avoiding large abrupt changes in the learning rate for each input variable. The default is True.

betas
[tuple , optional] Betas are used as for smoothing the path to the convergence also providing some momentum to cross a local minima or saddle point. The default is (0.9, 0.999).

eps
[float . optional.] eps is a term added to the denominator to improve numerical stability. The default is 0.000001.

lr
[float] The learning rate (lr) defines the adjustment in the weights of our network with respect to the loss gradient descent. The default is 0.001.

maximize
[float , optional] maximize the params based on the objective, instead of minimizing. The default is False.

weightDecay
[float , optional] weightDecay (L2 penalty) is a regularization technique applied to the weights of a neural network. The default is 0.0.

Returns

dict
The dictionary of the optimizer parameters. The dictionary contains the following keys and values: - "name" (str): The name of the optimizer - "amsgrad" (bool): - "betas" (tuple): - "eps" (float): - "lr" (float): - "maximize" (bool): - weightDecay (float):

`static Performance`(*actual*, *predicted*, *mantissa: int = 6*)

Computes regression model performance measures. This is to be used only with regression not with classification.

Parameters

actual
[list] The input list of actual values.

predicted
[list] The input list of predicted values.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
The dictionary containing the performance measures. The keys in the dictionary are: 'mae', 'mape', 'mse', 'r', 'r2', 'rmse'.

`static Show`(*data*, *labels*, *title='Training/Validation'*, *xTitle='Epochs'*, *xSpacing=1*, *yTitle='Accuracy and Loss'*, *ySpacing=0.1*, *useMarkers=False*, *chartType='Line'*, *width=950*, *height=500*, *backgroundColor='rgba(0,0,0,0)'*, *gridColor='lightgray'*, *marginLeft=0*, *marginRight=0*, *marginTop=40*, *marginBottom=0*, *renderer='notebook'*)

Shows the data in a plolty graph.

Parameters

data
[list] The data to display.

labels
[list] The labels to use for the data.

width
[int , optional] The desired width of the figure. The default is 950.

height
[int , optional] The desired height of the figure. The default is 500.

title
[str , optional] The chart title. The default is "Training and Testing Results".

xTitle
[str , optional] The X-axis title. The default is "Epochs".

xSpacing
[float , optional] The X-axis spacing. The default is 1.0.

yTitle
[str , optional] The Y-axis title. The default is "Accuracy and Loss".

ySpacing
[float , optional] The Y-axis spacing. The default is 0.1.

useMarkers
[bool , optional] If set to True, markers will be displayed. The default is False.

chartType
[str , optional] The desired type of chart. The options are "Line", "Bar", or "Scatter". It is case insensitive. The default is "Line".

backgroundColor
[str , optional] The desired background color. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'rgba(0,0,0,0)' (transparent).

gridColor
[str , optional] The desired grid color. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is 'lightgray'.

marginLeft
[int , optional] The desired left margin in pixels. The default is 0.

marginRight
[int , optional] The desired right margin in pixels. The default is 0.

marginTop
[int , optional] The desired top margin in pixels. The default is 40.

marginBottom
[int , optional] The desired bottom margin in pixels. The default is 0.

renderer
[str , optional] The desired plotly renderer. The default is "notebook".

Returns

None.

topologicpy.Shell module

class topologicpy.Shell.**Shell**

Bases: object

Methods

ByDisjointFaces(externalBoundary, faces[, ...])	Creates a shell from an input list of disjointed faces.
ByFaces(faces[, tolerance, silent])	Creates a shell from the input list of faces.
ByFacesCluster(cluster[, tolerance])	Creates a shell from the input cluster of faces.
ByThickenedWire(wire[, offsetA, offsetB, ...])	Creates a shell by thickening the input wire.
ByWires(wires[, triangulate, tolerance, silent])	Creates a shell by lofting through the input wires
ByWiresCluster(cluster[, triangulate, ...])	Creates a shell by lofting through the input cluster of wires
Circle([origin, radius, sides, fromAngle, ...])	Creates a circle.
Delaunay(vertices[, face, mantissa, tolerance])	Returns a delaunay partitioning of the input vertices.
Edges(shell)	Returns the edges of the input shell.
ExternalBoundary(shell[, tolerance])	Returns the external boundary of the input shell.
Faces(shell)	Returns the faces of the input shell.
HyperbolicParaboloidCircularDomain([origin, ...])	Creates a hyperbolic paraboloid with a circular domain.
HyperbolicParaboloidRectangularDomain([...])	Creates a hyperbolic paraboloid with a rectangular domain.
InternalBoundaries(shell[, tolerance])	Returns the internal boundaries (closed wires) of the input shell.
IsClosed(shell)	Returns True if the input shell is closed.
IsOnBoundary(shell, vertex[, tolerance])	Returns True if the input vertex is on the boundary of the input shell.
Paraboloid([origin, focalLength, width, ...])	Creates a paraboloid.
Pie([origin, radiusA, radiusB, sides, ...])	Creates a pie shape.
Planarize(shell[, origin, mantissa, tolerance])	Returns a planarized version of the input shell.
Rectangle([origin, width, length, uSides, ...])	Creates a rectangle.
RemoveCollinearEdges(shell[, angTolerance, ...])	Removes any collinear edges in the input shell.
Roof(face[, angle, epsilon, mantissa, tolerance])	Creates a hipped roof through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com>
SelfMerge(shell[, angTolerance, tolerance])	Creates a face by merging the faces of the input shell.
Simplify(shell[, simplifyBoundary, ...])	Simplifies the input shell edges based on the Douglas Peucker algorithm. See https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm
Skeleton([tolerance])	Creates a shell through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com>
Vertices(shell)	Returns the vertices of the input shell.
Voronoi(vertices[, face, mantissa, tolerance])	Returns a voronoi partitioning of the input face based on the input vertices.
Wires(shell)	Returns the wires of the input shell.

static ByDisjointFaces(*externalBoundary*, *faces*, *maximumGap: float = 0.5*, *mergeJunctions: bool = False*, *threshold: float = 0.5*, *uSides: int = 1*, *vSides: int = 1*, *transferDictionaries: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a shell from an input list of disjointed faces. THIS IS STILL EXPERIMENTAL

Parameters

externalBoundary
[topologic_core.Face] The input external boundary of the faces. This resembles a ribbon (face with hole) where its interior boundary touches the edges of the input list of faces.

faces
[list] The input list of faces.

maximumGap
[float , optional] The length of the maximum gap between the faces. The default is 0.5.

mergeJunctions
[bool , optional] If set to True, the interior junctions are merged into a single vertex. Otherwise, diagonal edges are added to resolve transitions between different gap distances.

threshold
[float , optional] The desired threshold under which vertices are merged into a single vertex. The default is 0.5.

uSides
[int , optional] The desired number of sides along the X axis for the grid that subdivides the input faces to aid in processing. The default is 1.

vSides
[int , optional] The desired number of sides along the Y axis for the grid that subdivides the input faces to aid in processing. The default is 1.

transferDictionaries
[bool, optional.] If set to True, the dictionaries in the input list of faces are transfered to the faces of the resulting shell. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created Shell.

`static ByFaces`(*faces: list*, *tolerance: float = 0.0001*, *silent=False*)

Creates a shell from the input list of faces.

Parameters

faces
[list] The input list of faces.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Shell
The created Shell.

`static ByFacesCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a shell from the input cluster of faces.

Parameters

cluster
[topologic_core.Cluster] The input cluster of faces.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created shell.

`static ByThickenedWire`(*wire*, *offsetA: float = 1.0*, *offsetB: float = 1.0*, *tolerance: float = 0.0001*)

Creates a shell by thickening the input wire. This method assumes the wire is manifold and planar.

Parameters

wire
[topologic_core.Wire] The input wire to be thickened.

offsetA
[float , optional] The desired offset to the exterior of the wire. The default is 1.0.

offsetB
[float , optional] The desired offset to the interior of the wire. The default is 1.0.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created shell.

`static ByWires`(*wires: list*, *triangulate: bool = True*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a shell by lofting through the input wires

Parameters

wires
[list] The input list of wires.

triangulate
[bool , optional] If set to True, the faces will be triangulated. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Shell
The creates shell.

static ByWiresCluster(*cluster*, *triangulate: bool = True*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a shell by lofting through the input cluster of wires

Parameters

wires
[topologic_core.Cluster] The input cluster of wires.

triangulate
[bool , optional] If set to True, the faces will be triangulated. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Shell
The creates shell.

static Circle(*origin=None*, *radius: float = 0.5*, *sides: int = 32*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a circle.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the circle. The default is None which results in the circle being placed at (0, 0, 0).

radius
[float , optional] The radius of the circle. The default is 0.5.

sides
[int , optional] The number of sides of the circle. The default is 32.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the circle. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the circle. The default is 360.

direction
[list , optional] The vector representing the up direction of the circle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the pie. This can be “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created circle.

`static Delaunay`(*vertices: list*, *face=None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a delaunay partitioning of the input vertices. The vertices must be coplanar. See https://en.wikipedia.org/wiki/Delaunay_triangulation.

Parameters

vertices
[list] The input list of vertices.

face
[topologic_core.Face , optional] The input face. If specified, the delaunay triangulation is clipped to the face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

shell
A shell representing the delaunay triangulation of the input vertices.

`static Edges`(*shell*) → list

Returns the edges of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

Returns

list
The list of edges.

`static ExternalBoundary`(*shell*, *tolerance: float = 0.0001*)

Returns the external boundary of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire or topologic_core.Cluster
The external boundary of the input shell. If the shell has holes, the return value will be a cluster of wires.

`static Faces`(*shell*) → list

Returns the faces of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

Returns

list
The list of faces.

static HyperbolicParaboloidCircularDomain(*origin=None*, *radius: float = 0.5*, *sides: int = 36*, *rings: int = 10*, *A: float = 2.0*, *B: float = -2.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a hyperbolic paraboloid with a circular domain. See https://en.wikipedia.org/wiki/Compactness_measure_of_a_shape

Parameters

origin
[topologic_core.Vertex , optional] The origin of the hyperbolic parabolid. If set to None, it will be placed at the (0, 0, 0) origin. The default is None.

radius
[float , optional] The desired radius of the hyperbolic paraboloid. The default is 0.5.

sides
[int , optional] The desired number of sides of the hyperbolic parabolid. The default is 36.

rings
[int , optional] The desired number of concentric rings of the hyperbolic parabolid. The default is 10.

A
[float , optional] The *A* constant in the equation z = A*x^2^ + B*y^2^. The default is 2.0.

B
[float , optional] The *B* constant in the equation z = A*x^2^ + B*y^2^. The default is -2.0.

direction
[list , optional] The vector representing the up direction of the hyperbolic paraboloid. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the circle. This can be “center”, “lowerleft”, “bottom”. It is case insensitive. The default is “center”.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created hyperbolic paraboloid.

static HyperbolicParaboloidRectangularDomain(*origin=None*, *llVertex=None*, *lrVertex=None*, *ulVertex=None*, *urVertex=None*, *uSides: int = 10*, *vSides: int = 10*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a hyperbolic paraboloid with a rectangular domain.

Parameters

origin
[topologic_core.Vertex , optional] The origin of the hyperbolic paraboloid. If set to None, it will be placed at the (0, 0, 0) origin. The default is None.

llVertex
[topologic_core.Vertex , optional] The lower left corner of the hyperbolic paraboloid. If set to None, it will be set to (-0.5, -0.5, -0.5).

lrVertex
[topologic_core.Vertex , optional] The lower right corner of the hyperbolic paraboloid. If set to None, it will be set to (0.5, -0.5, 0.5).

ulVertex
[topologic_core.Vertex , optional] The upper left corner of the hyperbolic paraboloid. If set to None, it will be set to (-0.5, 0.5, 0.5).

urVertex
[topologic_core.Vertex , optional] The upper right corner of the hyperbolic paraboloid. If set to None, it will be set to (0.5, 0.5, -0.5).

uSides
[int , optional] The number of segments along the X axis. The default is 10.

vSides
[int , optional] The number of segments along the Y axis. The default is 10.

direction
[list , optional] The vector representing the up direction of the hyperbolic paraboloid. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the hyperbolic paraboloid. This can be "center", "lowerleft", "bottom". It is case insensitive. The default is "center".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created hyperbolic paraboloid.

`static InternalBoundaries`(*shell*, *tolerance=0.0001*)

Returns the internal boundaries (closed wires) of the input shell. Internal boundaries are considered holes.

Parameters

shell
[topologic_core.Shell] The input shell.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of internal boundaries

`static IsClosed`(*shell*) → bool

Returns True if the input shell is closed. Returns False otherwise.

Parameters

shell
[topologic_core.Shell] The input shell.

Returns

bool
True if the input shell is closed. False otherwise.

`static IsOnBoundary`(*shell*, *vertex*, *tolerance: float = 0.0001*) → bool

Returns True if the input vertex is on the boundary of the input shell. Returns False otherwise. On the boundary is defined as being on the boundary of one of the shell's external or internal boundaries

Parameters

shell
[topologic_core.Shell] The input shell.

vertex
[topologic_core.Vertex] The input vertex.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
Returns True if the input vertex is inside the input shell. Returns False otherwise.

`static Paraboloid`(*origin=None*, *focalLength=0.125*, *width: float = 1*, *length: float = 1*, *uSides: int = 16*, *vSides: int = 16*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates a paraboloid. See https://en.wikipedia.org/wiki/Paraboloid

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the parabolic surface. The default is None which results in the parabolic surface being placed at (0, 0, 0).

focalLength
[float , optional] The focal length of the parabola. The default is 1.

width
[float , optional] The width of the parabolic surface. The default is 1.

length
[float , optional] The length of the parabolic surface. The default is 1.

uSides
[int , optional] The number of sides along the width. The default is 16.

vSides
[int , optional] The number of sides along the length. The default is 16.

direction
[list , optional] The vector representing the up direction of the parabolic surface. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the parabolic surface. This can be "bottom", "center", or "lowerleft". It is case insensitive. The default is "center".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional]

If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Shell
The created paraboloid.

`static Pie`(*origin=None*, *radiusA: float = 0.5*, *radiusB: float = 0.0*, *sides: int = 32*, *rings: int = 1*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a pie shape.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the pie. The default is None which results in the pie being placed at (0, 0, 0).

radiusA
[float , optional] The outer radius of the pie. The default is 0.5.

radiusB
[float , optional] The inner radius of the pie. The default is 0.25.

sides
[int , optional] The number of sides of the pie. The default is 32.

rings
[int , optional] The number of rings of the pie. The default is 1.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the pie. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the pie. The default is 360.

direction
[list , optional] The vector representing the up direction of the pie. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the pie. This can be “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created pie.

`static Planarize`(*shell*, *origin=None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a planarized version of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

origin
[topologic_core.Vertex , optional] The desired origin of the plane unto which the planar shell will be projected. If set to None, the centroid of the input shell will be chosen. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The planarized shell.

`static Rectangle`(*origin=None*, *width: float = 1.0*, *length: float = 1.0*, *uSides: int = 2*, *vSides: int = 2*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a rectangle.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the rectangle. The default is None which results in the rectangle being placed at (0, 0, 0).

width
[float , optional] The width of the rectangle. The default is 1.0.

length
[float , optional] The length of the rectangle. The default is 1.0.

uSides
[int , optional] The number of sides along the width. The default is 2.

vSides
[int , optional] The number of sides along the length. The default is 2.

direction
[list , optional] The vector representing the up direction of the rectangle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the rectangle. This can be “center”, or “lowerleft”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created shell.

`static RemoveCollinearEdges`(*shell*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Removes any collinear edges in the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Shell
The created shell without any collinear edges.

`static Roof`(*face*, *angle: float = 45*, *epsilon: float = 0.01*, *mantissa: int = 6*, *tolerance: float = 0.001*)

Creates a hipped roof through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com> This algorithm depends on the polyskel code which is included in the library. Polyskel code is found at: https://github.com/Botffy/polyskel

Parameters

face
[topologic_core.Face] The input face.

angle
[float , optioal] The desired angle in degrees of the roof. The default is 45.

epsilon
[float , optional] The desired epsilon (another form of tolerance for distance from plane). The default is 0.01. (This is set to a larger number as it was found to work better)

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.001. (This is set to a larger number as it was found to work better)

Returns

topologic_core.Shell
The created roof.

`static SelfMerge`(*shell*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Creates a face by merging the faces of the input shell. The shell must be planar within the input angular tolerance.

Parameters

shell
[topologic_core.Shell] The input shell.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Face
The created face.

`static Simplify`(*shell*, *simplifyBoundary: bool = True*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Simplifies the input shell edges based on the Douglas Peucker algorithm. See https://en.wikipedia.org/wiki/Ramer%E2%80%93Douglas%E2%80%93Peucker_algorithm Part of this code was contributed by gaoxipeng. See https://github.com/wassimj/topologicpy/issues/35

Parameters

shell
[topologic_core.Shell] The input shell.

simplifyBoundary
[bool , optional] If set to True, the external boundary of the shell will be simplified as well. Otherwise, it will not be simplified. The default is True.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001. Edges shorter than this length will be removed.

Returns

topologic_core.Shell
The simplified shell.

`Skeleton`(*tolerance: float = 0.001*)

Creates a shell through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com> This algorithm depends on the polyskel code which is included in the library. Polyskel code is found at: https://github.com/Botffy/polyskel

Parameters

face
[topologic_core.Face] The input face.

tolerance
[float , optional] The desired tolerance. The default is 0.001. (This is set to a larger number as it was found to work better)

Returns

topologic_core.Shell
The created straight skeleton.

`static Vertices`(*shell*) → list

Returns the vertices of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

Returns

list
The list of vertices.

`static Voronoi`(*vertices: list*, *face=None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a voronoi partitioning of the input face based on the input vertices. The vertices must be coplanar and within the face. See https://en.wikipedia.org/wiki/Voronoi_diagram.

Parameters

vertices
[list] The input list of vertices.

face
[topologic_core.Face , optional] The input face. If the face is not set an optimised bounding rectangle of the input vertices is used instead. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

shell
A shell representing the voronoi partitioning of the input face.

`static Wires`(*shell*) → list

Returns the wires of the input shell.

Parameters

shell
[topologic_core.Shell] The input shell.

Returns

list
The list of wires.

topologicpy.Speckle module

`class topologicpy.Speckle.Speckle`

Bases: `object`

Methods

BranchesByStream(client, stream)	**Parameters**
ClientByURL([url, token])	**Parameters**
CommitByID(commit_list, commit_id)	**Parameters**
CommitsByBranch(branch)	**Parameters**
Send(client, stream, branch, description, ...)	**Parameters**
SpeckleBranchByID(branch_list, branch_id)	**Parameters**
SpeckleCommitByURL(url, token)	**Parameters**
SpeckleCommitDelete(client, stream, commit, ...)	**Parameters**
SpeckleGlobalsByStream(client, stream)	**Parameters**
SpeckleSendObjects(client, stream, branch, ...)	**Parameters**
SpeckleStreamByID(stream_list, stream_id)	**Parameters**
SpeckleStreamByURL(url, token)	**Parameters**
StreamsByClient(item)	**Parameters**
get_name_from_IFC_name(name)	Function is used to get clean speckle element name

Object	
SpeckleObject	
TopologyBySpeckleObject	
mesh_to_speckle	
mesh_to_speckle_mesh	

static BranchesByStream(*client*, *stream*)

Parameters

client
[TYPE] DESCRIPTION.

stream
[TYPE] DESCRIPTION.

Returns

branches
[TYPE] DESCRIPTION.

`static ClientByURL(url='https://app.speckle.systems/', token=None)`

Parameters

url
[TYPE] DESCRIPTION.

token
[TYPE] DESCRIPTION.

Returns

client
[TYPE] DESCRIPTION.

`static CommitByID(commit_list, commit_id)`

Parameters

commit_list
[TYPE] DESCRIPTION.

commit_id
[TYPE] DESCRIPTION.

Returns

commit
[TYPE] DESCRIPTION.

`static CommitsByBranch(branch)`

Parameters

item
[TYPE] DESCRIPTION.

Returns

TYPE
DESCRIPTION.

`static Object(client, stream, branch, commit)`

`static Send(client, stream, branch, description, message, key, data, run)`

Parameters

client
[TYPE] DESCRIPTION.

stream
[TYPE] DESCRIPTION.

branch
[TYPE] DESCRIPTION.

description
[TYPE] DESCRIPTION.

message
[TYPE] DESCRIPTION.

key
[TYPE] DESCRIPTION.

data
[TYPE] DESCRIPTION.

run
[TYPE] DESCRIPTION.

Returns

commit
[TYPE] DESCRIPTION.

`static SpeckleBranchByID`(*branch_list*, *branch_id*)

Parameters

branch_list
[TYPE] DESCRIPTION.

branch_id
[TYPE] DESCRIPTION.

Returns

branch
[TYPE] DESCRIPTION.

`static SpeckleCommitByURL`(*url*, *token*)

Parameters

url
[TYPE] DESCRIPTION.

token
[TYPE] DESCRIPTION.

Returns

commit
[TYPE] DESCRIPTION.

`static SpeckleCommitDelete`(*client*, *stream*, *commit*, *confirm*)

Parameters

client
[TYPE] DESCRIPTION.

stream
[TYPE] DESCRIPTION.

commit
[TYPE] DESCRIPTION.

confirm
[TYPE] DESCRIPTION.

Returns

TYPE
DESCRIPTION.

`static SpeckleGlobalsByStream`(*client*, *stream*)

Parameters

client
[TYPE] DESCRIPTION.

stream
[TYPE] DESCRIPTION.

Returns

TYPE
DESCRIPTION.

`static SpeckleObject`(*client*, *stream*, *branch*, *commit*)

`static SpeckleSendObjects`(*client*, *stream*, *branch*, *description*, *message*, *key*, *data*, *run*)

Parameters

client
[TYPE] DESCRIPTION.

stream
[TYPE] DESCRIPTION.

branch
[TYPE] DESCRIPTION.

description
[TYPE] DESCRIPTION.

message
[TYPE] DESCRIPTION.

key
[TYPE] DESCRIPTION.

data
[TYPE] DESCRIPTION.

run
[TYPE] DESCRIPTION.

Returns

commit
[TYPE] DESCRIPTION.

`static SpeckleStreamByID`(*stream_list*, *stream_id*)

Parameters

stream_list
[TYPE] DESCRIPTION.

stream_id
[TYPE] DESCRIPTION.

Returns

stream
[TYPE] DESCRIPTION.

`static SpeckleStreamByURL(url, token)`

Parameters

url
[TYPE] DESCRIPTION.

token
[TYPE] DESCRIPTION.

Returns

stream
[TYPE] DESCRIPTION.

`static StreamsByClient(item)`

Parameters

item
[TYPE] DESCRIPTION.

Returns

TYPE
DESCRIPTION.

`static TopologyBySpeckleObject(obj)`

`static get_name_from_IFC_name(name)` → str
Function is used to get clean speckle element name

`static mesh_to_speckle(topology)` → Base

`static mesh_to_speckle_mesh(topology, mantissa: int = 6)` → Mesh

topologicpy.Sun module

`class topologicpy.Sun.Sun`
Bases: `object`

Methods

Altitude(latitude, longitude, date)	Returns the Altitude angle.
AutumnEquinox(latitude, year)	Returns the autumnal equinox date for the input latitude and year.
Azimuth(latitude, longitude, date)	Returns the Azimuth angle.
Diagram(latitude, longitude[, ...])	Returns the sun diagram based on the input parameters.
Edge(latitude, longitude, date[, origin, ...])	Returns the Sun as a vector.
PathByDate(latitude, longitude, date[, ...])	Returns the sun path based on the input parameters.
PathByHour(latitude, longitude, hour[, ...])	Returns the sun locations based on the input parameters.
Position(latitude, longitude, date[, ...])	Returns the Sun as a position ([X,Y,Z]).
SpringEquinox(latitude, year)	Returns the spring (vernal) equinox date for the input latitude and year.
SummerSolstice(latitude, year)	Returns the winter solstice date for the input latitude and year.
Sunrise(latitude, longitude, date)	Returns the Sunrise datetime.
Sunset(latitude, longitude, date)	Returns the Sunset datetime.
Vector(latitude, longitude, date[, north])	Returns the Sun as a vector.
Vertex(latitude, longitude, date[, origin, ...])	Returns the Sun as a vertex.
VerticesByDate(latitude, longitude, date[, ...])	Returns the Sun locations as vertices based on the input parameters.
VerticesByHour(latitude, longitude, hour[, ...])	Returns the sun locations based on the input parameters.
WinterSolstice(latitude[, year])	Returns the winter solstice date for the input latitude and year.

static Altitude(*latitude*, *longitude*, *date*)

Returns the Altitude angle. See https://en.wikipedia.org/wiki/Altitude.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

Returns

float
The altitude angle.

static AutumnEquinox(*latitude*, *year*)

Returns the autumnal equinox date for the input latitude and year. See https://en.wikipedia.org/wiki/September_equinox.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

year
[integer , optional] The input year. The default is the current year.

Returns

datetime
The datetime of the summer solstice

`static Azimuth`(*latitude*, *longitude*, *date*)

Returns the Azimuth angle. See https://en.wikipedia.org/wiki/Azimuth.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

Returns

float
The azimuth angle.

`static Diagram`(*latitude*, *longitude*, *minuteInterval=30*, *dayInterval=15*, *origin=None*, *radius=0.5*, *uSides=180*, *vSides=180*, *north=0*, *compass=False*, *shell=False*)

Returns the sun diagram based on the input parameters. See https://hyperfinearchitecture.com/how-to-read-sun-path-diagrams/.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

minuteInterval
[int , optional] The interval in minutes to compute the sun location for the date path. The default is 30.

dayInterval
[int , optional] The interval in days for the hourly path to compute the sun location. The default is 15.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

uSides
[int , optional] The number of sides to divide the diagram horizontally (along the azimuth). The default is 180.

vSides
[int , optional] The number of sides to divide the diagram paths vertically (along the altitude). The default is 180.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

compass
[bool , optional] If set to True, a compass (shell) is included. Othwerwise, it is is not.

shell
[bool , optional] If set to True, the total surface (shell) of the sun paths is incldued. Otherwise, it is not.

Returns

dict
A dictionary of the sun diagram shapes. The keys in this dictionary are: 'date_paths': These are the sun paths (wire) for the winter solstice, equinox, and summer solstice 'hourly_paths': These are the figure-8 (wire) for the sun location on the same hour on each selected day of the year. 'shell': This is the total surface (shell) of the sun paths. This is included only if the shell option is set to True. 'compass': This is the compass (shell) on the ground. It is made of 36 sides and 10 rings. This is included only if the compass option is set to True. 'center' : This is a cross-shape (wire) at the center of the diagram. This is included only if the compass option is set to True. 'ground' : This is a circle (face) on the ground. It is made of 36 sides. This is included only if the compass option is set to False.

`static Edge`(*latitude*, *longitude*, *date*, *origin=None*, *radius=0.5*, *north=0*)

Returns the Sun as a vector.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

topologic.Edge
The sun represented as an edge pointing from the location of the sun towards the origin.

`static PathByDate`(*latitude*, *longitude*, *date*, *startTime=None*, *endTime=None*, *interval=60*, *origin=None*, *radius=0.5*, *sides=None*, *north=0*)

Returns the sun path based on the input parameters.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

startTime
[datetime , optional] The desired start time to compute the sun location. If set to None, Sun.Sunrise is used. The default is None.

endTime
[datetime , optional] The desired end time to compute the sun location. If set to None, Sun.Sunset is used. The default is None.

interval
[int , optional] The interval in minutes to compute the sun location. The default is 60.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

sides
[int , optional] If set to None, the path is divided based on the interval. Otherwise, it is equally divided into the number of sides. The default is None.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

topologic.Wire
The sun path represented as a wire.

`static PathByHour`(*latitude*, *longitude*, *hour*, *startDay=1*, *endDay=365*, *interval=5*, *origin=None*, *radius=0.5*, *sides=None*, *north=0*)

Returns the sun locations based on the input parameters.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

hour
[datetime] The input hour.

startDay
[integer , optional] The desired start day of the year to compute the sun location. The default is 1.

endDay
[integer , optional] The desired end day of the year to compute the sun location. The default is 365.

interval
[int , optional] The interval in days to compute the sun location. The default is 5.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

sides
[int , optional] If set to None, the path is divided based on the interval. Otherwise, it is equally divided into the number of sides. The default is None.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

topologic.Wire
The sun path represented as a topologic wire.

`static Position`(*latitude*, *longitude*, *date*, *origin=None*, *radius=0.5*, *north=0*, *mantissa=6*)

Returns the Sun as a position ([X,Y,Z]).

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic.Vertex
The sun represented as a vertex.

`static SpringEquinox`(*latitude*, *year*)

Returns the spring (vernal) equinox date for the input latitude and year. See https://en.wikipedia.org/wiki/March_equinox.

Parameters

latitude
[float] The input latitude.

year
[integer , optional] The input year. The default is the current year.

Returns

datetime
The datetime of the summer solstice

`static SummerSolstice`(*latitude*, *year*)

Returns the winter solstice date for the input latitude and year. See https://en.wikipedia.org/wiki/Summer_solstice.

Parameters

latitude
[float] The input latitude.

year
[integer , optional] The input year. The default is the current year.

Returns

datetime
The datetime of the summer solstice

`static Sunrise`(*latitude*, *longitude*, *date*)

Returns the Sunrise datetime.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

Returns

datetime
The Sunrise datetime.

`static Sunset`(*latitude*, *longitude*, *date*)

Returns the Sunset datetime.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

Returns

datetime
The Sunset datetime.

`static Vector`(*latitude*, *longitude*, *date*, *north=0*)

Returns the Sun as a vector.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

list
The sun vector pointing from the location of the sun towards the origin.

`static Vertex`(*latitude*, *longitude*, *date*, *origin=None*, *radius=0.5*, *north=0*)

Returns the Sun as a vertex.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

topologic.Vertex
The sun represented as a vertex.

`static VerticesByDate`(*latitude*, *longitude*, *date*, *startTime=None*, *endTime=None*, *interval=60*, *origin=None*, *radius=0.5*, *north=0*)

Returns the Sun locations as vertices based on the input parameters.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

date
[datetime] The input datetime.

startTime
[datetime , optional] The desired start time to compute the sun location. If set to None, Sun.Sunrise is used. The default is None.

endTime
[datetime , optional] The desired end time to compute the sun location. If set to None, Sun.Sunset is used. The default is None.

interval
[int , optional] The interval in minutes to compute the sun location. The default is 60.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

list
The sun locations represented as a list of vertices.

`static VerticesByHour`(*latitude*, *longitude*, *hour*, *startDay=1*, *endDay=365*, *interval=5*, *origin=None*, *radius=0.5*, *north=0*)

Returns the sun locations based on the input parameters.

Parameters

latitude
[float] The input latitude. See https://en.wikipedia.org/wiki/Latitude.

longitude
[float] The input longitude. See https://en.wikipedia.org/wiki/Longitude.

hour
[datetime] The input hour.

startDay
[integer , optional] The desired start day to compute the sun location. The default is 1.

endDay
[integer , optional] The desired end day to compute the sun location. The default is 365.

interval
[int , optional] The interval in days to compute the sun location. The default is 5.

origin
[topologic.Vertex , optional] The desired origin of the world. If set to None, the origin will be set to (0,0,0). The default is None.

radius
[float , optional] The desired radius of the sun orbit. The default is 0.5.

north
[float, optional] The desired compass angle of the north direction. The default is 0 which points in the positive Y-axis direction.

Returns

list
The sun locations represented as a list of vertices.

`static WinterSolstice`(*latitude*, *year=None*)

Returns the winter solstice date for the input latitude and year. See https://en.wikipedia.org/wiki/Winter_solstice.

Parameters

latitude
[float] The input latitude.

year
[integer , optional] The input year. The default is the current year.

Returns

datetime
The datetime of the winter solstice

topologicpy.Topology module

`class topologicpy.Topology.MergingProcess`(*message_queue*, *sources*, *sinks*, *so_dicts*)

Bases: `Process`

Receive message from other processes and merging the result

Attributes

authkey
daemon
Return whether process is a daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

name
pid
Return identifier (PID) of process or *None* if it has yet to start

sentinel
Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.

Methods

close()	Close the Process object.
is_alive()	Return whether process is alive
join([timeout])	Wait until child process terminates
kill()	Terminate process; sends SIGKILL signal or uses TerminateProcess()
run()	Method to be run in sub-process; can be overridden in sub-class
start()	Start child process
terminate()	Terminate process; sends SIGTERM signal or uses TerminateProcess()

wait_message	

wait_message()

class topologicpy.Topology.**QueueItem**(*ID*, *sinkKeys*, *sinkValues*)

Bases: tuple

Attributes

ID
Alias for field number 0

sinkKeys
Alias for field number 1

sinkValues
Alias for field number 2

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

ID
Alias for field number 0

sinkKeys
Alias for field number 1

sinkValues
Alias for field number 2

class topologicpy.Topology.**SinkItem**(*ID*, *sink_str*)

Bases: tuple

Attributes

ID
Alias for field number 0

sink_str
Alias for field number 1

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

ID

Alias for field number 0

sink_str

Alias for field number 1

class topologicpy.Topology.**Topology**

Bases: object

Methods

AddApertures(topology, apertures[, ...])	Adds the input list of apertures to the input topology or to its subtopologies based on the input subTopologyType.
AddApertures_old(topology, apertures[, ...])	Adds the input list of apertures to the input topology or to its subtopologies based on the input subTopologyType.
AddContent(topology, contents[, ...])	Adds the input list of contents to the input topology or to its subtpologies based on the input subTopologyType.
AddDictionary(topology, dictionary)	Adds the input dictionary to the input topology.
AdjacentTopologies(topology, hostTopology[, ...])	Returns the topologies, as specified by the input topology type, adjacent to the input topology within the input host topology.
Analyze(topology)	Returns an analysis string that describes the input topology.
ApertureTopologies(topology[, subTopologyType])	Returns the aperture topologies of the input topology.
Apertures(topology[, subTopologyType])	Returns the apertures of the input topology.
BREPString(topology[, version])	Returns the BRep string of the input topology.
Boolean(topologyA, topologyB[, operation, ...])	Execute the input boolean operation type on the input operand topologies and return the result.
BoundingBox(topology[, optimize, axes, ...])	Returns a cell representing a bounding box of the input topology.
ByBIMFile(file[, guidKey, colorKey, ...])	Imports topologies from the input BIM (dotbimpy.file.File) file object.
ByBIMPath(path[, guidKey, colorKey, ...])	Imports topologies from the input BIM file.
ByBIMString(string[, guidKey, colorKey, ...])	Imports topologies from the input BIM file.
ByBREPFile(file)	Imports a topology from a BREP file.
ByBREPPath(path)	Imports a topology from a BREP file path.
ByBREPString(string)	Creates a topology from the input brep string
ByDXFFile(file[, sides])	Imports a list of topologies from a DXF file.
ByDXFPath(path[, sides])	Imports a list of topologies from a DXF file path.
ByGeometry([vertices, edges, faces, ...])	Create a topology by the input lists of vertices, edges, and faces.

continues on next page

Table 5 – continued from previous page

ByGeometry_old([vertices, edges, faces, ...])	Create a topology by the input lists of vertices, edges, and faces.
ByIFCFile(file[, includeTypes, ...])	Create a list of topologies by importing them from an IFC file.
ByIFCPath(path[, includeTypes, ...])	Create a topology by importing it from an IFC file path.
ByJSONDictionary(jsonDictionary[, tolerance])	Imports the topology from a JSON dictionary.
ByJSONFile(file[, tolerance])	Imports the topology from a JSON file.
ByJSONPath(path[, tolerance])	Imports the topology from a JSON file.
ByJSONString(string[, tolerance])	Imports the topology from a JSON string.
ByOBJFile(objFile[, mtlFile, defaultColor, ...])	Imports a topology from an OBJ file and an associated materials file.
ByOBJPath(objPath[, defaultColor, ...])	Imports a topology from an OBJ file path and an associated materials file.
ByOBJString(objString[, mtlString, ...])	Imports a topology from OBJ and MTL strings.
ByOCCTShape(occtShape)	Creates a topology from the input OCCT shape.
ByXYZFile(file[, frameIdKey, vertexIdKey])	Imports the topology from an XYZ file path.
ByXYZPath(path[, frameIdKey, vertexIdKey])	Imports the topology from an XYZ file path.
CellComplexes(topology)	Returns the cellcomplexes of the input topology.
Cells(topology)	Returns the cells of the input topology.
CenterOfMass(topology)	Returns the center of mass of the input topology.
Centroid(topology)	Returns the centroid of the vertices of the input topology.
Cleanup([topology])	Cleans up all resources in which are managed by topologic library.
ClusterByKeys(topologies, *keys[, silent])	Clusters the input list of topologies based on the input key or keys.
ClusterFaces(topology[, angTolerance, tolerance])	Clusters the faces of the input topology by their direction.
ClusterFaces_orig(topology[, angTolerance, ...])	Clusters the faces of the input topology by their direction.
Clusters(topology)	Returns the clusters of the input topology.
Contents(topology)	Returns the contents of the input topology
Contexts(topology)	Returns the list of contexts of the input topology
ConvexHull(topology[, mantissa, tolerance])	Creates a convex hull
Copy(topology[, deep])	Returns a copy of the input topology
Degree(topology, hostTopology)	Returns the number of immediate super topologies that use the input topology
Dictionary(topology)	Returns the dictionary of the input topology
Difference(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
Dimensionality(topology)	Returns the dimensionality of the input topology
Divide(topologyA, topologyB[, ...])	Divides the input topology by the input tool and places the results in the contents of the input topology.
Edges(topology)	Returns the edges of the input topology.
Explode(topology[, origin, scale, ...])	Explodes the input topology.
ExportToBIM(topologies, path[, overwrite, ...])	Exports the input topology to a BIM file.
ExportToBREP(topology, path[, overwrite, ...])	Exports the input topology to a BREP file.
ExportToDXF(path[, overwrite, mantissa])	Exports the input topology to a DXF file.
ExportToJSON(topologies, path[, overwrite])	Exports the input list of topologies to a JSON file.
ExportToOBJ(*topologies, path[, nameKey, ...])	Exports the input topology to a Wavefront OBJ file.

continues on next page

Table 5 – continued from previous page

ExternalBoundary(topology)	Returns the external boundary of the input topology.
Faces(topology)	Returns the faces of the input topology.
Filter(topologies[, topologyType, ...])	Filters the input list of topologies based on the input parameters.
Fix(topology[, topologyType, tolerance])	Attempts to fix the input topology to matched the desired output type.
Flatten(topology[, origin, direction, mantissa])	Flattens the input topology such that the input origin is located at the world origin and the input topology is rotated such that the input vector is pointed in the Up direction (see Vector.Up()).
Geometry(topology[, mantissa])	Returns the geometry (mesh data format) of the input topology as a dictionary of vertices, edges, and faces.
HighestType(topology)	Returns the highest topology type found in the input topology.
Impose(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
Imprint(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
InternalVertex(topology[, tolerance])	Returns a vertex guaranteed to be inside the input topology.
Intersect(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
IsInstance(topology, type)	Returns True if the input topology is an instance of the class specified by the input type string.
IsPlanar(topology[, mantissa, tolerance])	Returns True if all the vertices of the input topology are co-planar.
IsSame(topologyA, topologyB)	Returns True if the input topologies are the same topology.
JSONString(topologies[, mantissa])	Exports the input list of topologies to a JSON string
Merge(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
MergeAll(topologies[, tolerance])	Merge all the input topologies.
NonPlanarFaces(topology[, tolerance])	Returns any nonplanar faces in the input topology
OBJString(*topologies[, nameKey, colorKey, ...])	Exports the input topology to a Wavefront OBJ file.
OCCTShape(topology)	Returns the occt shape of the input topology.
OpenEdges(topology)	Returns the edges that border only one face.
OpenFaces(topology)	Returns the faces that border no cells.
OpenVertices(topology)	Returns the vertices that border only one edge.
Orient(topology[, origin, dirA, dirB, tolerance])	Orients the input topology such that the input such that the input dirA vector is parallel to the input dirB vector.
Place(topology[, originA, originB, mantissa])	Places the input topology at the specified location.
RemoveCollinearEdges(topology[, ...])	Removes the collinear edges of the input topology
RemoveContent(topology, contents)	Removes the input content list from the input topology
RemoveCoplanarFaces(topology[, epsilon, ...])	Removes coplanar faces in the input topology
RemoveEdges(topology[, edges, tolerance])	Removes the input list of faces from the input topology
RemoveFaces(topology[, faces, tolerance])	Removes the input list of faces from the input topology
RemoveFacesBySelectors(topology[, ...])	Removes faces that contain the input list of selectors (vertices) from the input topology
RemoveVertices(topology[, vertices, tolerance])	Removes the input list of vertices from the input topology

continues on next page

Table 5 – continued from previous page

ReplaceVertices(topology[, verticesA, ...])	Replaces the vertices in the first input list with the vertices in the second input list and rebuilds the input topology.
Rotate(topology[, origin, axis, angle, ...])	Rotates the input topology
RotateByEulerAngles(topology[, origin, ...])	Rotates the input topology using Euler angles (roll, pitch, yaw).
RotateByQuaternion(topology[, origin, ...])	Rotates the input topology using Quaternion rotations.
Scale(topology[, origin, x, y, z])	Scales the input topology
SelectSubTopology(topology, selector[, ...])	Returns the subtopology within the input topology based on the input selector and the subTopologyType.
SelfMerge(topology[, transferDictionaries, ...])	Self merges the input topology to return the most logical topology type given the input data.
SetDictionary(topology, dictionary[, silent])	Sets the input topology's dictionary to the input dictionary
SharedEdges(topologyA, topologyB)	Returns the shared edges between the two input topologies
SharedFaces(topologyA, topologyB)	Returns the shared faces between the two input topologies
SharedTopologies(topologyA, topologyB)	Returns the shared topologies between the two input topologies
SharedVertices(topologyA, topologyB)	Returns the shared vertices between the two input topologies
SharedWires(topologyA, topologyB)	Returns the shared wires between the two input topologies
Shells(topology)	Returns the shells of the input topology.
Show(*topologies[, opacityKey, ...])	Shows the input topology on screen.
Slice(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
SortBySelectors(topologies, selectors[, ...])	Sorts the input list of topologies according to the input list of selectors.
Spin(topology[, origin, triangulate, ...])	Spins the input topology around an axis to create a new topology.See https://en.wikipedia.org/wiki/Solid_of_revolution.
SubTopologies(topology[, subTopologyType])	Returns the subtopologies of the input topology as specified by the subTopologyType input string.
SuperTopologies(topology, hostTopology[, ...])	Returns the supertopologies connected to the input topology.
SymDif(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean().
SymmetricDifference(topologyA, topologyB[, ...])	See Topology.Boolean().
Taper(topology[, origin, ratioRange, ...])	Tapers the input topology.
TransferDictionaries(sources, sinks[, ...])	Transfers the dictionaries from the list of sources to the list of sinks.
TransferDictionariesBySelectors(topology, ...)	Transfers the dictionaries of the list of selectors to the subtopologies of the input topology based on the input parameters.
Transform(topology, matrix)	Transforms the input topology by the input 4X4 transformation matrix.
Translate(topology[, x, y, z])	Translates (moves) the input topology.
TranslateByDirectionDistance(topology[, ...])	Translates (moves) the input topology along the input direction by the specified distance.

continues on next page

Table 5 – continued from previous page

Triangulate(topology[, ...])	Triangulates the input topology.
Twist(topology[, origin, angleRange, ...])	Twists the input topology.
Type(topology)	Returns the type of the input topology.
TypeAsString(topology)	Returns the type of the input topology as a string.
TypeID([name])	Returns the type id of the input name string.
UUID(topology[, namespace])	Generate a UUID v5 based on the provided content and a fixed namespace.
Unflatten(topology[, origin, direction])	Unflattens the input topology such that the world origin is translated to the input origin and the input topology is rotated such that the Up direction (see Vector.Up()) is aligned with the input vector.
Union(topologyA, topologyB[, tranDict, ...])	See Topology.Boolean()
Vertices(topology)	Returns the vertices of the input topology.
View3D(*topologies[, uuid, nameKey, ...])	Sends the input topologies to 3dviewer.net.
Wires(topology)	Returns the wires of the input topology.
XOR(topologyA, topologyB[, tranDict, tolerance])	See Topology.Boolean().

SetSnapshot	
Snapshots	

static AddApertures(*topology*, *apertures*, *exclusive=False*, *subTopologyType=None*, *tolerance=0.001*)

Adds the input list of apertures to the input topology or to its subtopologies based on the input subTopologyType.

Parameters

topology
[topologic_core.Topology] The input topology.

apertures
[list] The input list of apertures.

exclusive
[bool , optional] If set to True, one (sub)topology will accept only one aperture. Otherwise, one (sub)topology can accept multiple apertures. The default is False.

subTopologyType
[string , optional] The subtopology type to which to add the apertures. This can be "cell", "face", "edge", or "vertex". It is case insensitive. If set to None, the apertures will be added to the input topology. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.001. This is larger than the usual 0.0001 as it seems to work better.

Returns

topologic_core.Topology
The input topology with the apertures added to it.

static AddApertures_old(*topology*, *apertures*, *exclusive=False*, *subTopologyType=None*, *tolerance=0.001*)

Adds the input list of apertures to the input topology or to its subtopologies based on the input subTopologyType.

Parameters

topology
[topologic_core.Topology] The input topology.

apertures
[list] The input list of apertures.

exclusive
[bool , optional] If set to True, one (sub)topology will accept only one aperture. Otherwise, one (sub)topology can accept multiple apertures. The default is False.

subTopologyType
[string , optional] The subtopology type to which to add the apertures. This can be "cell", "face", "edge", or "vertex". It is case insensitive. If set to None, the apertures will be added to the input topology. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.001. This is larger than the usual 0.0001 as it seems to work better.

Returns

topologic_core.Topology
The input topology with the apertures added to it.

`static AddContent`(*topology*, *contents*, *subTopologyType=None*, *tolerance=0.0001*)

Adds the input list of contents to the input topology or to its subtpologies based on the input subTopologyType.

Parameters

topology
[topologic_core.Topology] The input topology.

contents
[list or topologic_core.Topology] The input list of contents (of type topologic_core.Topology). A single topology is also accepted as input.

subTopologyType
[string , optional] The subtopology type to which to add the contents. This can be "cellcomplex", "cell", "shell", "face", "wire", "edge", or "vertex". It is case insensitive. If set to None, the contents will be added to the input topology. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The input topology with the contents added to it.

`static AddDictionary`(*topology*, *dictionary*)

Adds the input dictionary to the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

dictionary
[topologic_core.Dictionary] The input dictionary.

Returns

topologic_core.Topology
The input topology with the input dictionary added to it.

`static AdjacentTopologies`(*topology*, *hostTopology*, *topologyType=None*)

Returns the topologies, as specified by the input topology type, adjacent to the input topology within the input host topology.

Parameters

topology
[topologic_core.Topology] The input topology.

hostTopology
[topologic_core.Topology] The host topology in which to search.

topologyType
[str] The type of topology for which to search. This can be one of "vertex", "edge", "wire", "face", "shell", "cell", "cellcomplex". It is case-insensitive. If it is set to None, the type will be set to the same type as the input topology. The default is None.

Returns

adjacentTopologies
[list] The list of adjacent topologies.

`static Analyze`(*topology*)

Returns an analysis string that describes the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

str
The analysis string.

`static ApertureTopologies`(*topology*, *subTopologyType=None*)

Returns the aperture topologies of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

subTopologyType
[string , optional] The subtopology type from which to retrieve the apertures. This can be "cell", "face", "edge", or "vertex" or "all". It is case insensitive. If set to "all", then all apertures will be returned. If set to None, the apertures will be retrieved only from the input topology. The default is None.

Returns

list
The list of aperture topologies found in the input topology.

`static Apertures`(*topology*, *subTopologyType=None*)

Returns the apertures of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

subTopologyType
[string , optional] The subtopology type from which to retrieve the apertures. This can be "cell", "face", "edge", or "vertex" or "all". It is case insensitive. If set to "all", then all apertures will be returned. If set to None, the apertures will be retrieved only from the input topology. The default is None.

Returns

list
The list of apertures belonging to the input topology.

`static BREPString`(*topology*, *version=3*)

Returns the BRep string of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

version
[int , optional] The desired BRep version number. The default is 3.

Returns

str
The BREP string.

`static Boolean`(*topologyA*, *topologyB*, *operation='union'*, *tranDict=False*, *tolerance=0.0001*)

Execute the input boolean operation type on the input operand topologies and return the result. See https://en.wikipedia.org/wiki/Boolean_operation.

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

operation
[str , optional] The boolean operation. This can be one of "union", "difference", "intersect", "symdif", "merge", "slice", "impose", "imprint". It is case insensitive. The default is "union".

tranDict
[bool , optional] If set to True the dictionaries of the operands are merged and transferred to the result. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
the resultant topology.

`static BoundingBox`(*topology*, *optimize: int = 0*, *axes: str = 'xyz'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a cell representing a bounding box of the input topology. The returned cell contains a dictionary with keys "xrot", "yrot", and "zrot" that represents rotations around the X, Y, and Z axes. If applied in the order of Z, Y, X, the resulting box will become axis-aligned.

Parameters

topology
[topologic_core.Topology] The input topology.

optimize
[int , optional] If set to an integer from 1 (low optimization) to 10 (high optimization), the method will attempt to optimize the bounding box so that it reduces its surface area. The default is 0 which will result in an axis-aligned bounding box. The default is 0.

axes
[str , optional] Sets what axes are to be used for rotating the bounding box. This can be any permutation or substring of "xyz". It is not case sensitive. The default is "xyz".

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cell or topologic_core.Face
The bounding box of the input topology.

`static ByBIMFile`(*file*, *guidKey: str = 'guid'*, *colorKey: str = 'color'*, *typeKey: str = 'type'*, *defaultColor: list = [255, 255, 255, 1]*, *defaultType: str = 'Structure'*, *authorKey='author'*, *dateKey='date'*, *mantissa: int = 6*, *angTolerance: float = 0.001*, *tolerance: float = 0.0001*)

Imports topologies from the input BIM (dotbimpy.file.File) file object. See https://dotbim.net/

Parameters

file
[dotbimpy.file.File] The input dotbim file.

guidKey
[str , optional] The key to use to store the the guid of the topology. The default is "guid".

colorKey
[str , optional] The key to use to find the the color of the topology. The default is "color". If no color is found, the defaultColor parameter is used.

typeKey
[str , optional] The key to use to find the the type of the topology. The default is "type". If no type is found, the defaultType parameter is used.

defaultColor
[list , optional] The default color to use for the topology. The default is [255,255,255,1] which is opaque white.

defaultType
[str , optional] The default type to use for the topology. The default is "Structure".

authorKey
[str , optional] The key to use to store the author of the topology. The default is "author".

dateKey
[str , optional] The key to use to store the creation date of the topology. This should be in the formate "DD.MM.YYYY". If no date is found the date of import is used.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

angTolerance
[float , optional] The angle tolerance in degrees under which no rotation is carried out. The default is 0.001 degrees.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies

`static ByBIMPath`(*path*, *guidKey: str = 'guid'*, *colorKey: str = 'color'*, *typeKey: str = 'type'*, *defaultColor: list = [255, 255, 255, 1]*, *defaultType: str = 'Structure'*, *authorKey='author'*, *dateKey='date'*, *mantissa: int = 6*, *angTolerance: float = 0.001*, *tolerance: float = 0.0001*)

Imports topologies from the input BIM file. See https://dotbim.net/

Parameters

path :str
The path to the .bim file.

guidKey
[str , optional] The key to use to store the the guid of the topology. The default is "guid".

colorKey
[str , optional] The key to use to find the the color of the topology. The default is "color". If no color is found, the defaultColor parameter is used.

typeKey
[str , optional] The key to use to find the the type of the topology. The default is "type". If no type is found, the defaultType parameter is used.

defaultColor
[list , optional] The default color to use for the topology. The default is [255,255,255,1] which is opaque white.

defaultType
[str , optional] The default type to use for the topology. The default is "Structure".

authorKey
[str , optional] The key to use to store the author of the topology. The default is "author".

dateKey
[str , optional] The key to use to store the creation date of the topology. This should be in the formate "DD.MM.YYYY". If no date is found the date of import is used.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

angTolerance
[float , optional] The angle tolerance in degrees under which no rotation is carried out. The default is 0.001 degrees.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies

`static ByBIMString`(*string*, *guidKey: str = 'guid'*, *colorKey: str = 'color'*, *typeKey: str = 'type'*, *defaultColor: list = [255, 255, 255, 1]*, *defaultType: str = 'Structure'*, *authorKey: str = 'author'*, *dateKey: str = 'date'*, *mantissa: int = 6*, *angTolerance: float = 0.001*, *tolerance: float = 0.0001*)

Imports topologies from the input BIM file. See https://dotbim.net/

Parameters

string :str
The input dotbim str (in JSON format).

guidKey
[str , optional] The key to use to store the the guid of the topology. The default is "guid".

colorKey
[str , optional] The key to use to find the the color of the topology. The default is "color". If no color is found, the defaultColor parameter is used.

typeKey
[str , optional] The key to use to find the the type of the topology. The default is "type". If no type is found, the defaultType parameter is used.

defaultColor
[list , optional] The default color to use for the topology. The default is [255,255,255,1] which is opaque white.

defaultType
[str , optional] The default type to use for the topology. The default is "Structure".

authorKey
[str , optional] The key to use to store the author of the topology. The default is "author".

dateKey
[str , optional] The key to use to store the creation date of the topology. This should be in the formate "DD.MM.YYYY". If no date is found the date of import is used.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

angTolerance
[float , optional] The angle tolerance in degrees under which no rotation is carried out. The default is 0.001 degrees.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies

`static ByBREPFile`(*file*)

Imports a topology from a BREP file.

Parameters

file
[file object] The BREP file.

Returns

topologic_core.Topology
The imported topology.

static ByBREPPath(*path*)

Imports a topology from a BREP file path.

Parameters

path
[str] The path to the BREP file.

Returns

topologic_core.Topology
The imported topology.

static ByBREPString(*string*)

Creates a topology from the input brep string

Parameters

string
[str] The input brep string.

Returns

topologic_core.Topology
The created topology.

static ByDXFFile(*file*, *sides: int = 16*)

Imports a list of topologies from a DXF file. This is an experimental method with limited capabilities.

Parameters

file
[a DXF file object] The DXF file object.

sides
[int , optional] The desired number of sides of splines. The default is 16.

Returns

list
The list of imported topologies.

static ByDXFPath(*path*, *sides: int = 16*)

Imports a list of topologies from a DXF file path. This is an experimental method with limited capabilities.

Parameters

path
[str] The path to the DXF file.

sides
[int , optional] The desired number of sides of splines. The default is 16.

Returns

list
The list of imported topologies.

static ByGeometry(*vertices=[]*, *edges=[]*, *faces=[]*, *topologyType: str = None*, *tolerance: float = 0.0001*, *silent: bool = False*)

Create a topology by the input lists of vertices, edges, and faces.

Parameters

vertices
[list] The input list of vertices in the form of [x, y, z]

edges
[list , optional] The input list of edges in the form of [i, j] where i and j are vertex indices.

faces
[list , optional] The input list of faces in the form of [i, j, k, l, ...] where the items in the list are vertex indices. The face is assumed to be closed to the last vertex is connected to the first vertex automatically.

topologyType
[str , optional] The desired topology type. The options are: "Vertex", "Edge", "Wire", "Face", "Shell", "Cell", "CellComplex". If set to None, a "Cluster" will be returned. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topology
[topologic_core.Topology] The created topology. The topology will have a dictionary embedded in it that records the input attributes (color, id, lengthUnit, name, type)

`static ByGeometry_old`(*vertices=[]*, *edges=[]*, *faces=[]*, *color=[1.0, 1.0, 1.0, 1.0]*, *id=None*, *name=None*, *lengthUnit='METERS'*, *outputMode='default'*, *tolerance=0.0001*)

Create a topology by the input lists of vertices, edges, and faces.

Parameters

vertices
[list] The input list of vertices in the form of [x, y, z]

edges
[list , optional] The input list of edges in the form of [i, j] where i and j are vertex indices.

faces
[list , optional] The input list of faces in the form of [i, j, k, l, ...] where the items in the list are vertex indices. The face is assumed to be closed to the last vertex is connected to the first vertex automatically.

color
[list , optional] The desired color of the object in the form of [r, g, b, a] where the components are between 0 and 1 and represent red, blue, green, and alpha (transparency) respectively. The default is [1.0, 1.0, 1.0, 1.0].

id
[str , optional] The desired ID of the object. If set to None, an automatic uuid4 will be assigned to the object. The default is None.

name
[str , optional] The desired name of the object. If set to None, a default name "Topologic_[topology_type]" will be assigned to the object. The default is None.

lengthUnit
[str , optional] The length unit used for the object. The default is "METERS"

outputMode
[str , optional] The desired output mode of the object. This can be “wire”, “shell”, “cell”, “cellcomplex”, or “default”. It is case insensitive. The default is “default”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topology
[topologic_core.Topology] The created topology. The topology will have a dictionary embedded in it that records the input attributes (color, id, lengthUnit, name, type)

`static ByIFCFile`(*file*, *includeTypes=[]*, *excludeTypes=[]*, *transferDictionaries=False*, *removeCoplanarFaces=False*, *epsilon=0.0001*, *tolerance=0.0001*)

Create a list of topologies by importing them from an IFC file.

Parameters

file
[file object] The input IFC file.

includeTypes
[list , optional] The list of IFC object types to include. It is case insensitive. If set to an empty list, all types are included. The default is [].

excludeTypes
[list , optional] The list of IFC object types to exclude. It is case insensitive. If set to an empty list, no types are excluded. The default is [].

transferDictionaries
[bool , optional] If set to True, the dictionaries from the IFC file will be transferred to the topology. Otherwise, they won’t. The default is False.

removeCoplanarFaces
[bool , optional] If set to True, coplanar faces are removed. Otherwise they are not. The default is False.

epsilon
[float , optional]

The desired epsilon (another form of tolerance) for finding if two faces are coplanar. The default is 0.0001.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

——-

list
The created list of topologies.

`static ByIFCPath`(*path*, *includeTypes=[]*, *excludeTypes=[]*, *transferDictionaries=False*, *removeCoplanarFaces=False*, *epsilon=0.0001*, *tolerance=0.0001*)

Create a topology by importing it from an IFC file path.

Parameters

path
[str] The path to the IFC file.

includeTypes
[list , optional] The list of IFC object types to include. It is case insensitive. If set to an empty list, all types are included. The default is [].

excludeTypes
[list , optional] The list of IFC object types to exclude. It is case insensitive. If set to an empty list, no types are excluded. The default is [].

transferDictionaries
[bool , optional] If set to True, the dictionaries from the IFC file will be transferred to the topology. Otherwise, they won't. The default is False.

removeCoplanarFaces
[bool , optional] If set to True, coplanar faces are removed. Otherwise they are not. The default is False.

epsilon
[float , optional] The desired epsilon (another form of tolerance) for finding if two faces are coplanar. The default is 0.0001.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns
——-
list
The created list of topologies.

`static ByJSONDictionary`(*jsonDictionary*, *tolerance=0.0001*)

Imports the topology from a JSON dictionary.

Parameters

jsonDictionary
[dict] The input JSON dictionary.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies (Warning: the list could contain 0, 1, or many topologies, but this method will always return a list)

`static ByJSONFile`(*file*, *tolerance=0.0001*)

Imports the topology from a JSON file.

Parameters

file
[file object] The input JSON file.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies (Warning: the list could contain 0, 1, or many topologies, but this method will always return a list)

`static ByJSONPath`(*path*, *tolerance=0.0001*)

Imports the topology from a JSON file.

Parameters

path
[str] The file path to the json file.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies.

`static ByJSONString`(*string*, *tolerance=0.0001*)

Imports the topology from a JSON string.

Parameters

string
[str] The input JSON string.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of imported topologies (Warning: the list could contain 0, 1, or many topologies, but this method will always return a list)

`static ByOBJFile`(*objFile*, *mtlFile=None*, *defaultColor: list = [255, 255, 255]*, *defaultOpacity: float = 1.0*, *transposeAxes: bool = True*, *removeCoplanarFaces: bool = False*, *selfMerge: bool = True*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Imports a topology from an OBJ file and an associated materials file. This method is basic and does not support textures and vertex normals.

Parameters

objFile
[file object] The OBJ file.

mtlFile
[file object , optional] The MTL file. The default is None.

defaultColor
[list , optional] The default color to use if none is specified in the file. The default is [255, 255, 255] (white).

defaultOpacity
[float , optional] The default opacity to use if none is specified in the file. The default is 1.0 (fully opaque).

transposeAxes
[bool , optional] If set to True the Z and Y axes are transposed. Otherwise, they are not. The default is True.

removeCoplanarFaces
[bool , optional] If set to True, coplanar faces are merged. The default is True.

selfMerge
[bool , optional] If set to True, the faces of the imported topologies will each be self-merged to create higher-dimensional objects. Otherwise, they remain a cluster of faces. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001

Returns

list
The imported topologies.

`static ByOBJPath`(*objPath*, *defaultColor: list = [255, 255, 255]*, *defaultOpacity: float = 1.0*, *transposeAxes: bool = True*, *removeCoplanarFaces: bool = False*, *selfMerge: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Imports a topology from an OBJ file path and an associated materials file. This method is basic and does not support textures and vertex normals.

Parameters

objPath
[str] The path to the OBJ file.

defaultColor
[list , optional] The default color to use if none is specified in the file. The default is [255, 255, 255] (white).

defaultOpacity
[float , optional] The default opacity to use if none is specified in the file. The default is 1.0 (fully opaque).

transposeAxes
[bool , optional] If set to True the Z and Y axes are transposed. Otherwise, they are not. The default is True.

removeCoplanarFaces
[bool , optional] If set to True, coplanar faces are merged. The default is True.

selfMerge
[bool , optional] If set to True, the faces of the imported topologies will each be self-merged to create higher-dimensional objects. Otherwise, they remain a cluster of faces. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001

Returns

list
The imported topologies.

`static ByOBJString`(*objString: str*, *mtlString: str = None*, *defaultColor: list = [255, 255, 255]*, *defaultOpacity: float = 1.0*, *transposeAxes: bool = True*, *removeCoplanarFaces: bool = False*, *selfMerge: bool = False*, *mantissa=6*, *tolerance=0.0001*)

Imports a topology from OBJ and MTL strings.

Parameters

objString
[str] The string of the OBJ file.

mtlString
[str , optional] The string of the MTL file. The default is None.

defaultColor
[list , optional] The default color to use if none is specified in the string. The default is [255, 255, 255] (white).

defaultOpacity
[float , optional] The default opacity to use if none is specified in the string. The default is 1.0 (fully opaque).

transposeAxes
[bool , optional] If set to True the Z and Y axes are transposed. Otherwise, they are not. The default is True.

removeCoplanarFaces
[bool , optional] If set to True, coplanar faces are merged. The default is True.

selfMerge
[bool , optional] If set to True, the faces of the imported topologies will each be self-merged to create higher-dimensional objects. Otherwise, they remain a cluster of faces. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001

Returns

list
The imported topologies.

`static ByOCCTShape`(*occtShape*)

Creates a topology from the input OCCT shape. See https://dev.opencascade.org/doc/overview/html/occt_user_guides__modeling_data.html.

Parameters

occtShape
[topologic_core.TopoDS_Shape] The inoput OCCT Shape.

Returns

topologic_core.Topology
The created topology.

`static ByXYZFile`(*file*, *frameIdKey='id'*, *vertexIdKey='id'*)

Imports the topology from an XYZ file path. This is a very experimental method. While variants of the format exist, topologicpy reads XYZ files that conform to the following: An XYZ file can be made out of one or more frames. Each frame will be stored in a sepatate topologic cluster. First line: total number of vertices in the frame. This must be an integer. No other words or characters are allowed on this line. Second line: frame label. This is free text and will be stored in the dictionary of each frame (topologic_core.Cluster) All other lines: vertex_label, x, y, and z coordinates, separated by spaces, tabs, or commas. The vertex label must be one word with no spaces. It is stored in the dictionary of each vertex.

Example: 3 Frame 1 A 5.67 -3.45 2.61 B 3.91 -1.91 4 A 3.2 1.2 -12.3 4 Frame 2 B 5.47 -3.45 2.61 B 3.91 -1.93 3.1 A 3.2 1.2 -22.4 A 3.2 1.2 -12.3 3 Frame 3 A 5.67 -3.45 2.61 B 3.91 -1.91 4 C 3.2 1.2 -12.3

Parameters

file
[file object] The input XYZ file.

frameIdKey
[str , optional] The desired id key to use to store the ID of each frame in its dictionary. The default is "id".

vertexIdKey
[str , optional] The desired id key to use to store the ID of each point in its dictionary. The default is "id".

Returns

list
The list of frames (topologic_core.Cluster).

`static ByXYZPath`(*path*, *frameIdKey='id'*, *vertexIdKey='id'*)

Imports the topology from an XYZ file path. This is a very experimental method. While variants of the format exist, topologicpy reads XYZ files that conform to the following: An XYZ file can be made out of one or more frames. Each frame will be stored in a sepatate topologic cluster. First line: total number of vertices in the frame. This must be an integer. No other words or characters are allowed on this line. Second line: frame label. This is free text and will be stored in the dictionary of each frame (topologic_core.Cluster) All other lines: vertex_label, x, y, and z coordinates, separated by spaces, tabs, or commas. The vertex label must be one word with no spaces. It is stored in the dictionary of each vertex.

Example: 3 Frame 1 A 5.67 -3.45 2.61 B 3.91 -1.91 4 A 3.2 1.2 -12.3 4 Frame 2 B 5.47 -3.45 2.61 B 3.91 -1.93 3.1 A 3.2 1.2 -22.4 A 3.2 1.2 -12.3 3 Frame 3 A 5.67 -3.45 2.61 B 3.91 -1.91 4 C 3.2 1.2 -12.3

Parameters

path
[str] The input XYZ file path.

frameIdKey
[str , optional] The desired id key to use to store the ID of each frame in its dictionary. The default is "id".

vertexIdKey
[str , optional] The desired id key to use to store the ID of each point in its dictionary. The default is "id".

Returns

list
The list of frames (topologic_core.Cluster).

`static CellComplexes`(*topology*)

Returns the cellcomplexes of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of cellcomplexes.

`static Cells`(*topology*)

Returns the cells of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of cells.

`static CenterOfMass`(*topology*)

Returns the center of mass of the input topology. See https://en.wikipedia.org/wiki/Center_of_mass.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

topologic_core.Vertex
The center of mass of the input topology.

`static Centroid`(*topology*)

Returns the centroid of the vertices of the input topology. See https://en.wikipedia.org/wiki/Centroid.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

topologic_core.Vertex
The centroid of the input topology.

`static Cleanup`(*topology=None*)

Cleans up all resources in which are managed by topologic library. Use this to manage your application's memory consumption. USE WITH CARE. This methods deletes dictionaries, contents, and contexts

Parameters

topology
[topologic_core.Topology , optional] If specified the resources used by the input topology will be deleted. If not, ALL resources will be deleted.

Returns

topologic_core.Topology
The input topology, but with its resources deleted or None.

`static ClusterByKeys`(*topologies*, **keys*, *silent=False*)

Clusters the input list of topologies based on the input key or keys.

Parameters

topologies
[list] The input list of topologies.

keys
[str or list or comma-separated str input parameters] The key or keys in the topology's dictionary to use for clustering.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

list
A nested list of topologies where each element is a list of topologies with the same key values.

`static ClusterFaces`(*topology*, *angTolerance=2*, *tolerance=0.0001*)

Clusters the faces of the input topology by their direction.

Parameters

topology
[topologic_core.Topology] The input topology.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of clusters of faces where faces in the same cluster have the same direction.

`static ClusterFaces_orig`(*topology*, *angTolerance=0.1*, *tolerance=0.0001*)

Clusters the faces of the input topology by their direction.

Parameters

topology
[topologic_core.Topology] The input topology.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of clusters of faces where faces in the same cluster have the same direction.

`static Clusters`(*topology*)

Returns the clusters of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of clusters.

`static Contents`(*topology*)

Returns the contents of the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of contents of the input topology.

`static Contexts`(*topology*)

Returns the list of contexts of the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of contexts of the input topology.

`static ConvexHull`(*topology*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a convex hull

Parameters

topology
[topologic_core.Topology] The input Topology.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The created convex hull of the input topology.

`static Copy`(*topology*, *deep=False*)

Returns a copy of the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

deep
[bool , optional] If set to True, a deep copy will be performed (this is slow). Othwerwise, it will not. The default is False

Returns

topologic_core.Topology
A copy of the input topology.

`static Degree`(*topology*, *hostTopology*)

Returns the number of immediate super topologies that use the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

hostTopology
[topologic_core.Topology] The input host topology to which the input topology belongs

Returns

int
The degree of the topology (the number of immediate super topologies that use the input topology).

`static Dictionary`(*topology*)

Returns the dictionary of the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

topologic_core.Dictionary
The dictionary of the input topology.

`static Difference`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

`static Dimensionality`(*topology*)

Returns the dimensionality of the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

int
The dimensionality of the input topology.

`static Divide`(*topologyA*, *topologyB*, *transferDictionary=False*, *addNestingDepth=False*)

Divides the input topology by the input tool and places the results in the contents of the input topology.

Parameters

topologyA
[topologic_core.Topology] The input topology to be divided.

topologyB
[topologic_core.Topology] the tool used to divide the input topology.

transferDictionary
[bool , optional] If set to True the dictionary of the input topology is transferred to the divided topologies.

addNestingDepth
[bool , optional] If set to True the nesting depth of the division is added to the dictionaries of the divided topologies.

Returns

topologic_core.Topology
The input topology with the divided topologies added to it as contents.

`static Edges`(*topology*)

Returns the edges of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of edges.

`static Explode`(*topology*, *origin=None*, *scale: float = 1.25*, *typeFilter: str = None*, *axes: str = 'xyz'*, *transferDictionaries: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Explodes the input topology. See https://en.wikipedia.org/wiki/Exploded-view_drawing.

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The origin of the explosion. If set to None, the centroid of the input topology will be used. The default is None.

scale
[float , optional] The scale factor of the explosion. The default is 1.25.

typeFilter
[str , optional] The type of the subtopologies to explode. This can be any of "vertex", "edge", "face", or "cell". If set to None, a subtopology one level below the type of the input topology will be used. The default is None.

axes
[str , optional] Sets what axes are to be used for exploding the topology. This can be any permutation or substring of "xyz". It is not case sensitive. The default is "xyz".

transferDictionaries
[bool , optional] If set to True, the dictionaries of the original subTopologies are transferred to the exploded topologies. Otherwise, they are not. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Cluster
The exploded topology.

`static ExportToBIM`(*topologies*, *path: str*, *overwrite: bool = False*, *version: str = '1.0.0'*, *guidKey: str = 'guid'*, *colorKey: str = 'color'*, *typeKey: str = 'type'*, *defaultColor: list = [255, 255, 255, 1]*, *defaultType: str = 'Structure'*, *author: str = 'topologicpy'*, *date: str = None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Exports the input topology to a BIM file. See https://dotbim.net/

Parameters

topologies
[list or topologic_core.Topology] The input list of topologies or a single topology. The .bim format is restricted to triangulated meshes. No wires, edges, or vertices are supported.

path
[str] The input file path.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

version
[str , optional] The desired version number for the BIM file. The default is "1.0.0".

guidKey
[str , optional] The key to use to find the the guid of the topology. It is case insensitive. The default is "guid". If no guid is found, one is generated automatically.

colorKey
[str , optional] The key to use to find the the color of the topology. It is case insensitive. The default is "color". If no color is found, the defaultColor parameter is used.

typeKey
[str , optional] The key to use to find the the type of the topology. It is case insensitive. The default is "type". If no type is found, the defaultType parameter is used.

defaultColor
[list , optional] The default color to use for the topology. The default is [255,255,255,1] which is opaque white.

defaultType
[str , optional] The default type to use for the topology. The default is "Structure".

author
[str , optional] The author of the topology. The default is "topologicpy".

date
[str , optional] The creation date of the topology. This should be in the formate "DD.MM.YYYY". The default is None which uses the date of export.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the export operation is successful. False otherwise.

`static ExportToBREP`(*topology*, *path*, *overwrite=False*, *version=3*)

Exports the input topology to a BREP file. See https://dev.opencascade.org/doc/occt-6.7.0/overview/html/occt_brep_format.html.

Parameters

topology
[topologic_core.Topology] The input topology.

path
[str] The input file path.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

version
[int , optional] The desired version number for the BREP file. The default is 3.

Returns

bool
True if the export operation is successful. False otherwise.

`ExportToDXF`(*path: str*, *overwrite: bool = False*, *mantissa: int = 6*)

Exports the input topology to a DXF file. See https://en.wikipedia.org/wiki/AutoCAD_DXF. THe DXF version is 'R2010' This is experimental and only geometry is exported.

Parameters

topologies
[list or topologic_core.Topology] The input list of topologies. This can also be a single topologic_core.Topology.

path
[str] The input file path.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

bool
True if the export operation is successful. False otherwise.

`static ExportToJSON`(*topologies*, *path*, *overwrite=False*)

Exports the input list of topologies to a JSON file.

Parameters

topologies
[list] The input list of topologies.

path
[str] The path to the JSON file.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

Returns

bool
The status of exporting the JSON file. If True, the operation was successful. Otherwise, it was unsuccesful.

static ExportToOBJ(**topologies*, *path*, *nameKey='name'*, *colorKey='color'*, *opacityKey='opacity'*, *defaultColor=[256, 256, 256]*, *defaultOpacity=0.5*, *transposeAxes: bool = True*, *mode: int = 0*, *meshSize: float = None*, *overwrite: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Exports the input topology to a Wavefront OBJ file. This is very experimental and outputs a simple solid topology.

Parameters

topologies
[list or comma separated topologies] The input list of topologies.

path
[str] The input file path.

nameKey
[str , optional] The topology dictionary key under which to find the name of the topology. The default is "name".

colorKey
[str, optional] The topology dictionary key under which to find the color of the topology. The default is "color".

opacityKey
[str , optional] The topology dictionary key under which to find the opacity of the topology. The default is "opacity".

defaultColor
[list , optional] The default color to use if no color is stored in the topology dictionary. The default is [255,255, 255] (white).

defaultOpacity
[float , optional] The default opacity to use of no opacity is stored in the topology dictionary. This must be between 0 and 1. The default is 1 (fully opaque).

transposeAxes
[bool , optional] If set to True the Z and Y coordinates are transposed so that Y points "up"

mode
[int , optional] The desired mode of meshing algorithm (for triangulation). Several options are available: 0: Classic 1: MeshAdapt 3: Initial Mesh Only 5: Delaunay 6: Frontal-Delaunay 7: BAMG 8: Fontal-Delaunay for Quads 9: Packing of Parallelograms All options other than 0 (Classic) use the gmsh library. See https://gmsh.info/doc/texinfo/gmsh.html#Mesh-options WARNING: The options that use gmsh can be very time consuming and can create very heavy geometry.

meshSize
[float , optional] The desired size of the mesh when using the "mesh" option. If set to None, it will be calculated automatically and set to 10% of the overall size of the face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

Returns

bool
True if the export operation is successful. False otherwise.

`static ExternalBoundary`(*topology*)

Returns the external boundary of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

topologic_core.Topology
The external boundary of the input topology.

`static Faces`(*topology*)

Returns the faces of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of faces.

`static Filter`(*topologies*, *topologyType='any'*, *searchType='any'*, *key=None*, *value=None*)

Filters the input list of topologies based on the input parameters.

Parameters

topologies
[list] The input list of topologies.

topologyType
[str , optional] The type of topology to filter by. This can be one of “any”, “vertex”, “edge”, “wire”, “face”, “shell”, “cell”, “cellcomplex”, or “cluster”. It is case insensitive. The default is “any”.

searchType
[str , optional] The type of search query to conduct in the topology’s dictionary. This can be one of “any”, “equal to”, “contains”, “starts with”, “ends with”, “not equal to”, “does not contain”. The default is “any”.

key
[str , optional] The dictionary key to search within. The default is None which means it will filter by topology type only.

value
[str , optional] The value to search for at the specified key. The default is None which means it will filter by topology type only.

Returns

dict
A dictionary of filtered and other elements. The dictionary has two keys: - “filtered” The filtered topologies. - “other” The other topologies that did not meet the filter criteria.

`static Fix`(*topology*, *topologyType: str = 'CellComplex'*, *tolerance: float = 0.0001*)

Attempts to fix the input topology to matched the desired output type.

Parameters

topology
[topologic_core.Topology] The input topology

topologyType
[str , optional] The desired output topology type. This must be one of "vertex", "edge", "wire", "face", "shell", "cell", "cellcomplex", "cluster". It is case insensitive. The default is "CellComplex"

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The output topology in the desired type.

`static Flatten`(*topology*, *origin=None*, *direction: list = [0, 0, 1]*, *mantissa: int = 6*)

Flattens the input topology such that the input origin is located at the world origin and the input topology is rotated such that the input vector is pointed in the Up direction (see Vector.Up()).

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The input origin. If set to None, The object's centroid will be used to place the world origin. The default is None.

direction
[list , optional] The input direction vector. The input topology will be rotated such that this vector is pointed in the positive Z axis.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic_core.Topology
The flattened topology.

`static Geometry`(*topology*, *mantissa: int = 6*)

Returns the geometry (mesh data format) of the input topology as a dictionary of vertices, edges, and faces.

Parameters

topology
[topologic_core.Topology] The input topology.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
A dictionary containing the vertices, edges, and faces data. The keys found in the dictionary are "vertices", "edges", and "faces".

`static HighestType`(*topology*)

Returns the highest topology type found in the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

int
The highest type found in the input topology.

`static Impose`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

`static Imprint`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

`static InternalVertex`(*topology*, *tolerance: float = 0.0001*)

Returns a vertex guaranteed to be inside the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

tolerance
[float , ptional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Vertex
A vertex guaranteed to be inside the input topology.

`static Intersect`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

`static IsInstance`(*topology*, *type: str*)

Returns True if the input topology is an instance of the class specified by the input type string.

Parameters

topology
[topologic_core.Topology] The input topology.

type
[string] The topology type. This can be one of: “Vertex” “Edge” “Wire” “Face” “Shell” “Cell” “CellComplex” “Cluster” “Topology” “Graph” “Aperture” “Dictionary” “Context”

Returns

bool
True if the input topology is an instance of the class defined by the input type string. False otherwise.

`static IsPlanar`(*topology*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns True if all the vertices of the input topology are co-planar. Returns False otherwise.

Parameters

topology
[topologic_core.Topology] The input topology.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if all the vertices of the input topology are co-planar. False otherwise.

`static IsSame`(*topologyA*, *topologyB*)

Returns True if the input topologies are the same topology. Returns False otherwise.

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

Returns

bool
True of the input topologies are the same topology. False otherwise.

`static JSONString`(*topologies*, *mantissa: int = 6*)

Exports the input list of topologies to a JSON string

Parameters

topologies
[list or topologic_core.Topology] The input list of topologies or a single topology.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

bool
The status of exporting the JSON file. If True, the operation was successful. Otherwise, it was unsuccesful.

`static Merge`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

`static MergeAll`(*topologies*, *tolerance=0.0001*)

Merge all the input topologies.

Parameters

topologies
[list] The list of input topologies.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The resulting merged Topology

`static NonPlanarFaces(topology, tolerance=0.0001)`

Returns any nonplanar faces in the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of nonplanar faces.

`static OBJString(*topologies, nameKey='name', colorKey='color', opacityKey='opacity', defaultColor=[256, 256, 256], defaultOpacity=0.5, transposeAxes: bool = True, mode: int = 0, meshSize: float = None, mantissa: int = 6, tolerance: float = 0.0001)`

Exports the input topology to a Wavefront OBJ file. This is very experimental and outputs a simple solid topology.

Parameters

topologies
[list or comma separated topologies] The input list of topologies.

nameKey
[str , optional] The topology dictionary key under which to find the name of the topology. The default is "name".

colorKey
[str, optional] The topology dictionary key under which to find the color of the topology. The default is "color".

opacityKey
[str , optional] The topology dictionary key under which to find the opacity of the topology. The default is "opacity".

defaultColor
[list , optional] The default color to use if no color is stored in the topology dictionary. The default is [255,255, 255] (white).

defaultOpacity
[float , optional] The default opacity to use of no opacity is stored in the topology dictionary. This must be between 0 and 1. The default is 1 (fully opaque).

transposeAxes
[bool , optional] If set to True the Z and Y coordinates are transposed so that Y points "up"

mode
[int , optional] The desired mode of meshing algorithm (for triangulation). Several options are available: 0: Classic 1: MeshAdapt 3: Initial Mesh Only 5: Delaunay 6: Frontal-Delaunay 7: BAMG 8: Fontal-Delaunay for Quads 9: Packing of Parallelograms All options other than 0 (Classic) use the gmsh library. See https://gmsh.info/doc/texinfo/gmsh.html#Mesh-options WARNING: The options that use gmsh can be very time consuming and can create very heavy geometry.

meshSize
[float , optional] The desired size of the mesh when using the "mesh" option. If set to None, it will be calculated automatically and set to 10% of the overall size of the face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
Return the OBJ and MTL strings as a list.

`static OCCTShape`(*topology*)

Returns the occt shape of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

topologic_core.TopoDS_Shape
The OCCT Shape.

`static OpenEdges`(*topology*)

Returns the edges that border only one face.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of open edges.

`static OpenFaces`(*topology*)

Returns the faces that border no cells.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of open edges.

`static OpenVertices`(*topology*)

Returns the vertices that border only one edge.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of open edges.

static Orient(*topology*, *origin=None*, *dirA=[0, 0, 1]*, *dirB=[0, 0, 1]*, *tolerance=0.0001*)

Orients the input topology such that the input such that the input dirA vector is parallel to the input dirB vector.

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The input origin. If set to None, The object's centroid will be used to locate the input topology. The default is None.

dirA
[list , optional] The first input direction vector. The input topology will be rotated such that this vector is parallel to the input dirB vector. The default is [0, 0, 1].

dirB
[list , optional] The target direction vector. The input topology will be rotated such that the input dirA vector is parallel to this vector. The default is [0, 0, 1].

tolerance
[float , optional] The desired tolerance. The default is 0.0001

Returns

topologic_core.Topology
The flattened topology.

static Place(*topology*, *originA=None*, *originB=None*, *mantissa: int = 6*)

Places the input topology at the specified location.

Parameters

topology
[topologic_core.Topology] The input topology.

originA
[topologic_core.Vertex , optional] The old location to use as the origin of the movement. If set to None, the centroid of the input topology is used. The default is None.

originB
[topologic_core.Vertex , optional] The new location at which to place the topology. If set to None, the world origin (0, 0, 0) is used. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6

Returns

topologic_core.Topology
The placed topology.

static RemoveCollinearEdges(*topology*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*, *silent: bool = False*)

Removes the collinear edges of the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Topology
The input topology with the collinear edges removed.

`static RemoveContent`(*topology*, *contents*)

Removes the input content list from the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

contentList
[list] The input list of contents.

Returns

topologic_core.Topology
The input topology with the input list of contents removed.

`static RemoveCoplanarFaces`(*topology*, *epsilon=0.01*, *tolerance=0.0001*)

Removes coplanar faces in the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

epsilon
[float , optional] The desired epsilon (another form of tolerance) for finding if two faces are coplanar. The default is 0.01.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The input topology with coplanar faces merged into one face.

`static RemoveEdges`(*topology*, *edges=[]*, *tolerance=0.0001*)

Removes the input list of faces from the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

edges
[list] The input list of edges.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The input topology with the input list of edges removed.

`static RemoveFaces`(*topology*, *faces=[]*, *tolerance=0.0001*)

Removes the input list of faces from the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

faces
[list] The input list of faces.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The input topology with the input list of faces removed.

`static RemoveFacesBySelectors`(*topology*, *selectors=[]*, *tolerance=0.0001*)

Removes faces that contain the input list of selectors (vertices) from the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

selectors
[list] The input list of selectors (vertices).

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The input topology with the identified faces removed.

`static RemoveVertices`(*topology*, *vertices=[]*, *tolerance=0.0001*)

Removes the input list of vertices from the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

vertices
[list] The input list of vertices.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The input topology with the input list of vertices removed.

static ReplaceVertices(*topology*, *verticesA: list = []*, *verticesB: list = []*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Replaces the vertices in the first input list with the vertices in the second input list and rebuilds the input topology. The two lists must be of the same length.

Parameters

topology
[topologic_core.Topology] The input topology.

verticesA
[list] The first input list of vertices.

verticesB
[list] The second input list of vertices.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The new topology.

static Rotate(*topology*, *origin=None*, *axis: list = [0, 0, 1]*, *angle: float = 0*, *angTolerance: float = 0.001*, *tolerance: float = 0.0001*)

Rotates the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The origin (center) of the rotation. If set to None, the world origin (0, 0, 0) is used. The default is None.

axis
[list , optional] The vector representing the axis of rotation. The default is [0, 0, 1] which equates to the Z axis.

angle
[float , optional] The angle of rotation in degrees. The default is 0.

angTolerance
[float , optional] The angle tolerance in degrees under which no rotation is carried out. The default is 0.001 degrees.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The rotated topology.

static RotateByEulerAngles(*topology*, *origin=None*, *roll: float = 0*, *pitch: float = 0*, *yaw: float = 0*, *angTolerance: float = 0.001*, *tolerance: float = 0.0001*)

Rotates the input topology using Euler angles (roll, pitch, yaw). See https://en.wikipedia.org/wiki/Aircraft_principal_axes

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The origin (center) of the rotation. If set to None, the world origin (0, 0, 0) is used. The default is None.

roll
[float , optional] The rotation angle in degrees around the X-axis. The default is 0.

pitch = float , optional
The rotation angle in degrees around the Y-axis. The default is 0.

yaw = float , optional
The rotation angle in degrees around the Z-axis. The default is 0.

angTolerance
[float , optional] The angle tolerance in degrees under which no rotation is carried out. The default is 0.001 degrees.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The rotated topology.

`static RotateByQuaternion`(*topology*, *origin=None*, *quaternion: list = [0, 0, 0, 1]*, *angTolerance: float = 0.001*, *tolerance: float = 0.0001*)

Rotates the input topology using Quaternion rotations. See https://en.wikipedia.org/wiki/Quaternion

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The origin (center) of the rotation. If set to None, the world origin (0, 0, 0) is used. The default is None.

quaternion
[list or numpy array of size 4] The input Quaternion list. It should be in the form [x, y, z, w].

angTolerance
[float , optional] The angle tolerance in degrees under which no rotation is carried out. The default is 0.001 degrees.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The rotated topology.

`static Scale`(*topology*, *origin=None*, *x=1*, *y=1*, *z=1*)

Scales the input topology

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The origin (center) of the scaling. If set to None, the world origin (0, 0, 0) is used. The default is None.

x
[float , optional] The 'x' component of the scaling factor. The default is 1.

y
[float , optional] The 'y' component of the scaling factor. The default is 1.

z
[float , optional] The 'z' component of the scaling factor. The default is 1..

Returns

topologic_core.Topology
The scaled topology.

`static SelectSubTopology`(*topology*, *selector*, *subTopologyType='vertex'*)

Returns the subtopology within the input topology based on the input selector and the subTopologyType.

Parameters

topology
[topologic_core.Topology] The input topology.

selector
[topologic_core.Vertex] A vertex located on the desired subtopology.

subTopologyType
[str , optional.] The desired subtopology type. This can be of "vertex", "edge", "wire", "face", "shell", "cell", or "cellcomplex". It is case insensitive. The default is "vertex".

Returns

topologic_core.Topology
The selected subtopology.

`static SelfMerge`(*topology*, *transferDictionaries: bool = False*, *tolerance: float = 0.0001*)

Self merges the input topology to return the most logical topology type given the input data.

Parameters

topology
[topologic_core.Topology] The input topology.

tolerance
[float , optional] The desired tolerance. The default is 0.0001

Returns

topologic_core.Topology
The self-merged topology.

`static SetDictionary`(*topology*, *dictionary*, *silent=False*)

Sets the input topology's dictionary to the input dictionary

Parameters

topology
[topologic_core.Topology] The input topology.

dictionary
[topologic_core.Dictionary] The input dictionary.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Topology
The input topology with the input dictionary set in it.

`static SetSnapshot`(*topology*, *snapshot=None*, *timestamp=None*, *key='timestamp'*, *silent=False*)

`static SharedEdges`(*topologyA*, *topologyB*)

Returns the shared edges between the two input topologies

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

Returns

list
The list of shared edges.

`static SharedFaces`(*topologyA*, *topologyB*)

Returns the shared faces between the two input topologies

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

Returns

list
The list of shared faces.

`static SharedTopologies`(*topologyA*, *topologyB*)

Returns the shared topologies between the two input topologies

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

Returns

dict
A dictionary with the list of vertices, edges, wires, and faces. The keys are "vertices", "edges", "wires", and "faces".

`static SharedVertices`(*topologyA*, *topologyB*)

Returns the shared vertices between the two input topologies

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

Returns

list
The list of shared vertices.

`static SharedWires`(*topologyA*, *topologyB*)

Returns the shared wires between the two input topologies

Parameters

topologyA
[topologic_core.Topology] The first input topology.

topologyB
[topologic_core.Topology] The second input topology.

Returns

list
The list of shared wires.

`static Shells`(*topology*)

Returns the shells of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of shells.

static Show(*topologies*, *opacityKey='opacity'*, *showVertices=True*, *vertexSize=1.1*, *vertexSizeKey=None*, *vertexColor='black'*, *vertexColorKey=None*, *vertexLabelKey=None*, *showVertexLabel=False*, *vertexGroupKey=None*, *vertexGroups=[]*, *vertexMinGroup=None*, *vertexMaxGroup=None*, *showVertexLegend=False*, *vertexLegendLabel='Topology Vertices'*, *vertexLegendRank=1*, *vertexLegendGroup=1*, *showEdges=True*, *edgeWidth=1*, *edgeWidthKey=None*, *edgeColor='black'*, *edgeColorKey=None*, *edgeLabelKey=None*, *showEdgeLabel=False*, *edgeGroupKey=None*, *edgeGroups=[]*, *edgeMinGroup=None*, *edgeMaxGroup=None*, *showEdgeLegend=False*, *edgeLegendLabel='Topology Edges'*, *edgeLegendRank=2*, *edgeLegendGroup=2*, *showFaces=True*, *faceOpacity=0.5*, *faceOpacityKey=None*, *faceColor='#FAFAFA'*, *faceColorKey=None*, *faceLabelKey=None*, *faceGroupKey=None*, *faceGroups=[]*, *faceMinGroup=None*, *faceMaxGroup=None*, *showFaceLegend=False*, *faceLegendLabel='Topology Faces'*, *faceLegendRank=3*, *faceLegendGroup=3*, *intensityKey=None*, *intensities=[]*, *width=950*, *height=500*, *xAxis=False*, *yAxis=False*, *zAxis=False*, *axisSize=1*, *backgroundColor='rgba(0,0,0,0)'*, *marginLeft=0*, *marginRight=0*, *marginTop=20*, *marginBottom=0*, *camera=[-1.25, -1.25, 1.25]*, *center=[0, 0, 0]*, *up=[0, 0, 1]*, *projection='perspective'*, *renderer='notebook'*, *showScale=False*, *cbValues=[]*, *cbTicks=5*, *cbX=-0.15*, *cbWidth=15*, *cbOutlineWidth=0*, *cbTitle=''*, *cbSubTitle=''*, *cbUnits=''*, *colorScale='Viridis'*, *sagitta=0*, *absolute=False*, *sides=8*, *angle=0*, *mantissa=6*, *tolerance=0.0001*, *silent=False*)

Shows the input topology on screen.

Parameters

topologies
[topologic_core.Topology or list] The input topology. This must contain faces and or edges. If the input is a list, a cluster is first created

opacityKey
[str , optional] The key under which to find the opacity of the topology. The default is "opacity".

showVertices
[bool , optional] If set to True the vertices will be drawn. Otherwise, they will not be drawn. The default is True.

vertexSize
[float , optional] The desired size of the vertices. The default is 1.1.

vertexSizeKey
[str , optional] The key under which to find the size of the vertex. The default is None.

vertexColor
[str , optional] The desired color of the output vertices. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

vertexColorKey
[str , optional] The key under which to find the color of the vertex. The default is None.

vertexLabelKey
[str , optional] The dictionary key to use to display the vertex label. The default is None.

showVertexLabels
[bool , optional] If set to True, the vertex labels are shown permenantely on screen. Otherwise, they are not. The default is False.

vertexGroupKey
[str , optional] The dictionary key to use to display the vertex group. The default is None.

vertexGroups
[list , optional] The list of vertex groups against which to index the color of the vertex. The default is [].

vertexMinGroup
[int or float , optional] For numeric vertexGroups, vertexMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the minimum value in vertexGroups. The default is None.

edgeMaxGroup
[int or float , optional] For numeric vertexGroups, vertexMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the vertexGroupKey. If set to None, it is set to the maximum value in vertexGroups. The default is None.

showVertexLegend
[bool, optional] If set to True, the legend for the vertices of this topology is shown. Otherwise, it isn't. The default is False.

vertexLegendLabel
[str , optional] The legend label string used to identify vertices. The default is "Topology Vertices".

vertexLegendRank
[int , optional] The legend rank order of the vertices of this topology. The default is 1.

vertexLegendGroup
[int , optional] The number of the vertex legend group to which the vertices of this topology belong. The default is 1.

showEdges
[bool , optional] If set to True the edges will be drawn. Otherwise, they will not be drawn. The default is True.

edgeWidth
[float , optional] The desired thickness of the output edges. The default is 1.

edgeColor
[str , optional] The desired color of the output edges. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "black".

edgeColorKey
[str , optional] The key under which to find the color of the edge. The default is None.

edgeWidthKey
[str , optional] The key under which to find the width of the edge. The default is None.

edgeLabelKey
[str , optional] The dictionary key to use to display the edge label. The default is None.

showEdgeLabels
[bool , optional] If set to True, the edge labels are shown permenantely on screen. Otherwise, they are not. The default is False.

edgeGroupKey
[str , optional] The dictionary key to use to display the edge group. The default is None.

edgeGroups
[list , optional] The list of edge groups against which to index the color of the edge. The default is [].

edgeMinGroup
[int or float , optional] For numeric edgeGroups, edgeMinGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the edgeGroupKey. If set to None, it is set to the minimum value in edgeGroups. The default is None.

edgeMaxGroup
[int or float , optional] For numeric edgeGroups, edgeMaxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the edgeGroupKey. If set to None, it is set to the maximum value in edgeGroups. The default is None.

showEdgeLegend
[bool, optional] If set to True, the legend for the edges of this topology is shown. Otherwise, it isn't. The default is False.

edgeLegendLabel
[str , optional] The legend label string used to identify edges. The default is "Topology Edges".

edgeLegendRank
[int , optional] The legend rank order of the edges of this topology. The default is 2.

edgeLegendGroup
[int , optional] The number of the edge legend group to which the edges of this topology belong. The default is 2.

showFaces
[bool , optional] If set to True the faces will be drawn. Otherwise, they will not be drawn. The default is True.

faceOpacity
[float , optional] The desired opacity of the output faces (0=transparent, 1=opaque). The default is 0.5.

faceOpacityKey
[str , optional] The key under which to find the opacity of the face. The default is None.

faceColor
[str , optional] The desired color of the output faces. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "#FAFAFA".

faceColorKey
[str , optional] The key under which to find the color of the face. The default is None.

faceLabelKey
[str , optional] The dictionary key to use to display the face label. The default is None.

faceGroupKey
[str , optional] The dictionary key to use to display the face group. The default is None.

faceGroups
[list , optional] The list of face groups against which to index the color of the face. This can bhave numeric or string values. This should match the type of value associated with the faceGroupKey. The default is [].

faceMinGroup
[int or float , optional] For numeric faceGroups, minGroup is the desired minimum value for the scaling of colors. This should match the type of value associated with the faceGroupKey. If set to None, it is set to the minimum value in faceGroups. The default is None.

faceMaxGroup
[int or float , optional] For numeric faceGroups, maxGroup is the desired maximum value for the scaling of colors. This should match the type of value associated with the faceGroupKey. If set to None, it is set to the maximum value in faceGroups. The default is None.

showFaceLegend
[bool, optional] If set to True, the legend for the faces of this topology is shown. Otherwise, it isn't. The default is False.

faceLegendLabel
[str , optional] The legend label string used to idenitfy edges. The default is "Topology Faces".

faceLegendRank
[int , optional] The legend rank order of the faces of this topology. The default is 3.

faceLegendGroup
[int , optional] The number of the face legend group to which the faces of this topology belong. The default is 3.

width
[int , optional] The width in pixels of the figure. The default value is 950.

height
[int , optional] The height in pixels of the figure. The default value is 950.

xAxis
[bool , optional] If set to True the x axis is drawn. Otherwise it is not drawn. The default is False.

yAxis
[bool , optional] If set to True the y axis is drawn. Otherwise it is not drawn. The default is False.

zAxis
[bool , optional] If set to True the z axis is drawn. Otherwise it is not drawn. The default is False.

backgroundColor
[str , optional] The desired color of the background. This can be any plotly color string and may be specified as: - A hex string (e.g. '#ff0000') - An rgb/rgba string (e.g. 'rgb(255,0,0)') - An hsl/hsla string (e.g. 'hsl(0,100%,50%)') - An hsv/hsva string (e.g. 'hsv(0,100%,100%)') - A named CSS color. The default is "rgba(0,0,0,0)".

marginLeft
[int , optional] The size in pixels of the left margin. The default value is 0.

marginRight
[int , optional] The size in pixels of the right margin. The default value is 0.

marginTop
[int , optional] The size in pixels of the top margin. The default value is 20.

marginBottom
[int , optional] The size in pixels of the bottom margin. The default value is 0.

camera
[list , optional] The desired location of the camera). The default is [-1.25, -1.25, 1.25].

center
[list , optional] The desired center (camera target). The default is [0, 0, 0].

up
[list , optional] The desired up vector. The default is [0, 0, 1].

projection
[str , optional] The desired type of projection. The options are "orthographic" or "perspective". It is case insensitive. The default is "perspective"

renderer
[str , optional] The desired renderer. See Plotly.Renderers(). If set to None, the code will attempt to discover the most suitable renderer. The default is None.

intensityKey
[str , optional] If not None, the dictionary of each vertex is searched for the value associated with the intensity key. This value is then used to color-code the vertex based on the colorScale. The default is None.

intensities
[list , optional] The list of intensities against which to index the intensity of the vertex. The default is [].

showScale
[bool , optional] If set to True, the colorbar is shown. The default is False.

cbValues
[list , optional] The input list of values to use for the colorbar. The default is [].

cbTicks
[int , optional] The number of ticks to use on the colorbar. The default is 5.

cbX
[float , optional] The x location of the colorbar. The default is -0.15.

cbWidth
[int , optional] The width in pixels of the colorbar. The default is 15

cbOutlineWidth
[int , optional] The width in pixels of the outline of the colorbar. The default is 0.

cbTitle
[str , optional] The title of the colorbar. The default is "".

cbSubTitle
[str , optional] The subtitle of the colorbar. The default is "".

cbUnits: str , optional
The units used in the colorbar. The default is ""

colorScale
[str , optional] The desired type of plotly color scales to use (e.g. "viridis", "plasma"). The default is "viridis". For a full list of names, see https://plotly.com/python/builtin-colorscales/.

mantissa
[int , optional] The desired length of the mantissa for the values listed on the colorbar. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

None

`static Slice`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)
See Topology.Boolean().

`static Snapshots`(*topology*, *key='timestamp'*, *start=None*, *end=None*, *silent=False*)

`static SortBySelectors`(*topologies*, *selectors*, *exclusive=False*, *tolerance=0.0001*)
Sorts the input list of topologies according to the input list of selectors.

Parameters

topologies
[list] The input list of topologies.

selectors
[list] The input list of selectors (vertices).

exclusive
[bool , optional] If set to True only one selector can be used to select on topology. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dict
A dictionary containing the list of sorted and unsorted topologies. The keys are "sorted" and "unsorted".

`static Spin`(*topology*, *origin=None*, *triangulate: bool = True*, *direction: list = [0, 0, 1]*, *angle: float = 360*, *sides: int = 16*, *tolerance: float = 0.0001*, *silent: bool = False*)
Spins the input topology around an axis to create a new topology.See https://en.wikipedia.org/wiki/Solid_of_revolution.

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex] The origin (center) of the spin.

triangulate
[bool , optional] If set to True, the result will be triangulated. The default is True.

direction
[list , optional] The vector representing the direction of the spin axis. The default is [0, 0, 1].

angle
[float , optional] The angle in degrees for the spin. The default is 360.

sides
[int , optional] The desired number of sides. The default is 16.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The spun topology.

`static SubTopologies`(*topology*, *subTopologyType='vertex'*)

Returns the subtopologies of the input topology as specified by the subTopologyType input string.

Parameters

topology
[topologic_core.Topology] The input topology.

subTopologyType
[str , optional] The requested subtopology type. This can be one of "vertex", "edge", "wire", "face", "shell", "cell", "cellcomplex", "cluster". It is case insensitive. The default is "vertex".

Returns

list
The list of subtopologies.

`static SuperTopologies`(*topology*, *hostTopology*, *topologyType: str = None*) → list

Returns the supertopologies connected to the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

hostTopology
[topologic_core.Topology] The host to topology in which to search for ther supertopologies.

topologyType
[str , optional] The topology type to search for. This can be any of "edge", "wire", "face", "shell", "cell", "cellcomplex", "cluster". It is case insensitive. If set to None, the immediate supertopology type is searched for. The default is None.

Returns

list
The list of supertopologies connected to the input topology.

`static SymDif`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

`static SymmetricDifference`(*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*)

See Topology.Boolean().

static Taper(*topology*, *origin=None*, *ratioRange: list = [0, 1]*, *triangulate: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Tapers the input topology. This method tapers the input geometry along its Z-axis based on the ratio range input.

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The desired origin for tapering. If not specified, the centroid of the input topology is used. The tapering will use the X, Y coordinates of the specified origin, but will use the Z of the point being tapered. The default is None.

ratioRange
[list , optional] The desired ratio range. This will specify a linear range from bottom to top for tapering the vertices. 0 means no tapering, and 1 means maximum (inward) tapering. Negative numbers mean that tapering will be outwards.

triangulate
[bool , optional] If set to true, the input topology is triangulated before tapering. Otherwise, it will not be traingulated. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. Vertices will not be moved if the calculated distance is at or less than this tolerance.

Returns

topologic_core.Topology
The tapered topology.

static TransferDictionaries(*sources*, *sinks*, *tolerance=0.0001*, *numWorkers=None*)

Transfers the dictionaries from the list of sources to the list of sinks.

Parameters

sources
[list] The list of topologies from which to transfer the dictionaries.

sinks
[list] The list of topologies to which to transfer the dictionaries.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

numWorkers
[int, optional] Number of workers run in parallel to process. The default is None which sets the number to twice the number of CPU cores.

Returns

dict
Returns a dictionary with the lists of sources and sinks. The keys are “sinks” and “sources”.

static TransferDictionariesBySelectors(*topology*, *selectors*, *tranVertices=False*, *tranEdges=False*, *tranFaces=False*, *tranCells=False*, *tolerance=0.0001*, *numWorkers=None*)

Transfers the dictionaries of the list of selectors to the subtopologies of the input topology based on the input parameters.

Parameters

topology
[topologic_core.Topology] The input topology.

selectors
[list] The list of input selectors from which to transfer the dictionaries.

tranVertices
[bool , optional] If True transfer dictionaries to the vertices of the input topology.

tranEdges
[bool , optional] If True transfer dictionaries to the edges of the input topology.

tranFaces
[bool , optional] If True transfer dictionaries to the faces of the input topology.

tranCells
[bool , optional] If True transfer dictionaries to the cells of the input topology.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

numWorkers
[int , optional] Number of workers run in parallel to process. If you set it to 1, no parallel processing will take place. The default is None which causes the algorithm to use twice the number of cpu cores in the host computer.

Returns

——-

Topology
The input topology with the dictionaries transferred to its subtopologies.

`static Transform`(*topology*, *matrix*)

Transforms the input topology by the input 4X4 transformation matrix.

Parameters

topology
[topologic_core.Topology] The input topology.

matrix
[list] The input 4x4 transformation matrix.

Returns

topologic_core.Topology
The transformed topology.

`static Translate`(*topology*, *x=0*, *y=0*, *z=0*)

Translates (moves) the input topology.

Parameters

topology
[topologic_core.topology] The input topology.

x
[float , optional] The x translation value. The default is 0.

y
[float , optional] The y translation value. The default is 0.

z
[float , optional] The z translation value. The default is 0.

Returns

topologic_core.Topology
The translated topology.

`static TranslateByDirectionDistance`(*topology*, *direction: list = [0, 0, 0]*, *distance: float = 0*)

Translates (moves) the input topology along the input direction by the specified distance.

Parameters

topology
[topologic_core.topology] The input topology.

direction
[list , optional] The direction vector in which the topology should be moved. The default is [0, 0, 0]

distance
[float , optional] The distance by which the toplogy should be moved. The default is 0.

Returns

topologic_core.Topology
The translated topology.

`static Triangulate`(*topology*, *transferDictionaries: bool = False*, *mode: int = 0*, *meshSize: float = None*, *tolerance: float = 0.0001*)

Triangulates the input topology.

Parameters

topology
[topologic_core.Topology] The input topologgy.

transferDictionaries
[bool , optional] If set to True, the dictionaries of the faces in the input topology will be transferred to the created triangular faces. The default is False.

mode
[int , optional] The desired mode of meshing algorithm. Several options are available: 0: Classic 1: MeshAdapt 3: Initial Mesh Only 5: Delaunay 6: Frontal-Delaunay 7: BAMG 8: Fontal-Delaunay for Quads 9: Packing of Parallelograms All options other than 0 (Classic) use the gmsh library. See https://gmsh.info/doc/texinfo/gmsh.html#Mesh-options WARNING: The options that use gmsh can be very time consuming and can create very heavy geometry.

meshSize
[float , optional] The desired size of the mesh when using the "mesh" option. If set to None, it will be calculated automatically and set to 10% of the overall size of the face.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The triangulated topology.

static Twist(*topology*, *origin=None*, *angleRange: list = [45, 90]*, *triangulate: bool = False*, *mantissa: int = 6*)

Twists the input topology. This method twists the input geometry along its Z-axis based on the degree range input.

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The desired origin for tapering. If not specified, the centroid of the input topology is used. The tapering will use the X, Y coordinates of the specified origin, but will use the Z of the point being tapered. The default is None.

angleRange
[list , optional] The desired angle range in degrees. This will specify a linear range from bottom to top for twisting the vertices. positive numbers mean a clockwise rotation.

triangulate
[bool , optional] If set to true, the input topology is triangulated before tapering. Otherwise, it will not be traingulated. The default is False.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic_core.Topology
The twisted topology.

static Type(*topology*)

Returns the type of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

int
The type of the input topology.

static TypeAsString(*topology*)

Returns the type of the input topology as a string.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

str
The type of the topology as a string.

static TypeID(*name: str = None*) → int

Returns the type id of the input name string.

Parameters

name
[str , optional]

The input class name string. This could be one of:
"vertex", "edge", "wire", "face", "shell", "cell", "cellcomplex", "cluster", "aperture", "context", "dictionary", "graph", "topology"

It is case insensitive. The default is None.

Returns

int
The type id of the input topologyType string.

`static UUID(`*topology*, *namespace='topologicpy'*`)`

Generate a UUID v5 based on the provided content and a fixed namespace.

Parameters

topology
[topologic_core.Topology] The input topology

namespace
[str , optional] The base namescape to use for generating the UUID

Returns

UUID
The uuid of the input topology.

`static Unflatten(`*topology*, *origin=None*, *direction=[0, 0, 1]*`)`

Unflattens the input topology such that the world origin is translated to the input origin and the input topology is rotated such that the Up direction (see Vector.Up()) is aligned with the input vector.

Parameters

topology
[topologic_core.Topology] The input topology.

origin
[topologic_core.Vertex , optional] The input origin. If set to None, The object's centroid will be used to translate the world origin. The default is None.

vector
[list , optional] The input direction vector. The input topology will be rotated such that this vector is pointed in the positive Z axis.

Returns

topologic_core.Topology
The flattened topology.

`static Union(`*topologyA*, *topologyB*, *tranDict=False*, *tolerance=0.0001*`)`

See Topology.Boolean()

`static Vertices(`*topology*`)`

Returns the vertices of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of vertices.

static View3D(**topologies*, *uuid=None*, *nameKey='name'*, *colorKey='color'*, *opacityKey='opacity'*, *defaultColor=[256, 256, 256]*, *defaultOpacity=0.5*, *transposeAxes: bool = True*, *mode: int = 0*, *meshSize: float = None*, *overwrite: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Sends the input topologies to 3dviewer.net. The topologies must be 3D meshes.

Parameters

topologies
[list or comma separated topologies] The input list of topologies.

uuid
[UUID , optional] The UUID v5 to use to identify these topologies. The default is a UUID based on the topologies themselves.

nameKey
[str , optional] The topology dictionary key under which to find the name of the topology. The default is "name".

colorKey
[str, optional] The topology dictionary key under which to find the color of the topology. The default is "color".

opacityKey
[str , optional] The topology dictionary key under which to find the opacity of the topology. The default is "opacity".

defaultColor
[list , optional] The default color to use if no color is stored in the topology dictionary. The default is [255,255, 255] (white).

defaultOpacity
[float , optional] The default opacity to use of no opacity is stored in the topology dictionary. This must be between 0 and 1. The default is 1 (fully opaque).

transposeAxes
[bool , optional] If set to True the Z and Y coordinates are transposed so that Y points "up"

mode
[int , optional] The desired mode of meshing algorithm (for triangulation). Several options are available: 0: Classic 1: MeshAdapt 3: Initial Mesh Only 5: Delaunay 6: Frontal-Delaunay 7: BAMG 8: Fontal-Delaunay for Quads 9: Packing of Parallelograms All options other than 0 (Classic) use the gmsh library. See https://gmsh.info/doc/texinfo/gmsh.html#Mesh-options WARNING: The options that use gmsh can be very time consuming and can create very heavy geometry.

meshSize
[float , optional] The desired size of the mesh when using the "mesh" option. If set to None, it will be calculated automatically and set to 10% of the overall size of the face.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

overwrite
[bool , optional] If set to True the ouptut file will overwrite any pre-existing file. Otherwise, it won't. The default is False.

Returns

bool
True if the export operation is successful. False otherwise.

`static Wires(topology)`

Returns the wires of the input topology.

Parameters

topology
[topologic_core.Topology] The input topology.

Returns

list
The list of wires.

`static XOR(topologyA, topologyB, tranDict=False, tolerance=0.0001)`

See Topology.Boolean().

`class topologicpy.Topology.WorkerProcess(message_queue, sources, sinks, so_dicts, tolerance=0.0001)`

Bases: `Process`

Transfers the dictionaries from a subset of sources to the list of sinks.

Attributes

authkey
daemon
Return whether process is a daemon

exitcode
Return exit code of process or *None* if it has yet to stop

ident
Return identifier (PID) of process or *None* if it has yet to start

name
pid
Return identifier (PID) of process or *None* if it has yet to start

sentinel
Return a file descriptor (Unix) or handle (Windows) suitable for waiting for process termination.

Methods

close()	Close the Process object.
is_alive()	Return whether process is alive
join([timeout])	Wait until child process terminates
kill()	Terminate process; sends SIGKILL signal or uses TerminateProcess()
run()	Method to be run in sub-process; can be overridden in sub-class
start()	Start child process
terminate()	Terminate process; sends SIGTERM signal or uses TerminateProcess()

run()

Method to be run in sub-process; can be overridden in sub-class

class topologicpy.Topology.**WorkerProcessPool**(*num_workers*, *message_queue*, *sources*, *sinks*, *so_dicts*, *tolerance=0.0001*)

Bases: object

Create and manage a list of Worker processes. Each worker process transfers the dictionaries from a subset of sources to the list of sinks.

Methods

join	
startProcesses	
stopProcesses	

join()

startProcesses()

stopProcesses()

topologicpy.Vector module

class topologicpy.Vector.**Vector**(*iterable=()*, */*)

Bases: list

Methods

Add(vectorA, vectorB)	Adds the two input vectors.
Angle(vectorA, vectorB[, mantissa])	Returns the angle in degrees between the two input vectors
Average(vectors)	Returns the average vector of the input vectors.
AzimuthAltitude(vector[, mantissa])	Returns a dictionary of azimuth and altitude angles in degrees for the input vector.

continues on next page

Table 6 – continued from previous page

Bisect(vectorA, vectorB)	Compute the bisecting vector of two input vectors.
ByAzimuthAltitude(azimuth, altitude[, ...])	Returns the vector specified by the input azimuth and altitude angles.
ByCoordinates(x, y, z)	Creates a vector by the specified x, y, z inputs.
ByVertices(vertices[, normalize, mantissa])	Creates a vector by the specified input list of vertices.
CompassAngle(vectorA, vectorB[, mantissa, ...])	Returns the horizontal compass angle in degrees between the two input vectors.
Coordinates(vector[, outputType, mantissa])	Returns the coordinates of the input vector.
Cross(vectorA, vectorB[, mantissa, tolerance])	Returns the cross product of the two input vectors.
Dot(vectorA, vectorB[, mantissa])	Returns the dot product of the two input vectors which is a measure of how much they are aligned.
Down()	Returns the vector representing the *down* direction.
East()	Returns the vector representing the *east* direction.
IsAntiParallel(vectorA, vectorB)	Returns True if the input vectors are anti-parallel.
IsCollinear(vectorA, vectorB)	Returns True if the input vectors are collinear (parallel or anti-parallel).
IsParallel(vectorA, vectorB)	Returns True if the input vectors are parallel.
IsSame(vectorA, vectorB[, tolerance])	Returns True if the input vectors are the same.
Length(vector[, mantissa])	Returns the length of the input vector.
Magnitude(vector[, mantissa])	Returns the magnitude of the input vector.
Multiply(vector, magnitude[, tolerance])	Multiplies the input vector by the input magnitude.
Normalize(vector)	Returns a normalized vector of the input vector.
North()	Returns the vector representing the *north* direction.
NorthEast()	Returns the vector representing the *northeast* direction.
NorthWest()	Returns the vector representing the *northwest* direction.
Reverse(vector)	Returns a reverse vector of the input vector.
SetMagnitude(vector, magnitude)	Sets the magnitude of the input vector to the input magnitude.
South()	Returns the vector representing the *south* direction.
SouthEast()	Returns the vector representing the *southeast* direction.
SouthWest()	Returns the vector representing the *southwest* direction.
Subtract(vectorA, vectorB)	Subtracts the second input vector from the first input vector.
Sum(vectors)	Returns the sum vector of the input vectors.
TransformationMatrix(vectorA, vectorB)	Returns the transformation matrix needed to align vectorA with vectorB.
Up()	Returns the vector representing the up direction.
West()	Returns the vector representing the *west* direction.
XAxis()	Returns the vector representing the XAxis ([1, 0, 0])
YAxis()	Returns the vector representing the YAxis ([0, 1, 0])
ZAxis()	Returns the vector representing the ZAxis ([0, 0, 1])
append(object, /)	Append object to the end of the list.
clear(/)	Remove all items from list.
copy(/)	Return a shallow copy of the list.
count(value, /)	Return number of occurrences of value.
extend(iterable, /)	Extend list by appending elements from the iterable.
index(value[, start, stop])	Return first index of value.

continues on next page

Table 6 – continued from previous page

insert(index, object, /)	Insert object before index.
pop([index])	Remove and return item at index (default last).
remove(value, /)	Remove first occurrence of value.
reverse(/)	Reverse *IN PLACE*.
sort(*[, key, reverse])	Sort the list in ascending order and return None.

static Add(*vectorA*, *vectorB*)

Adds the two input vectors.

Parameters

vectorA
[list] The first vector.

vectorB
[list] The second vector.

Returns

list
The sum vector of the two input vectors.

static Angle(*vectorA*, *vectorB*, *mantissa: int = 6*)

Returns the angle in degrees between the two input vectors

Parameters

vectorA
[list] The first vector.

vectorB
[list] The second vector.

mantissa
[int, optional] The length of the desired mantissa. The default is 6.

Returns

float
The angle in degrees between the two input vectors.

static Average(*vectors: list*)

Returns the average vector of the input vectors.

Parameters

vectors
[list] The input list of vectors.

Returns

list
The average vector of the input list of vectors.

static AzimuthAltitude(*vector*, *mantissa: int = 6*)

Returns a dictionary of azimuth and altitude angles in degrees for the input vector. North is assumed to be the positive Y axis [0, 1, 0]. Up is assumed to be the positive Z axis [0, 0, 1]. Azimuth is calculated in a counter-clockwise fashion from North where 0 is North, 90 is East, 180 is South, and 270 is West. Altitude is calculated in a counter-clockwise fashion where -90 is straight down (negative Z axis), 0 is in the XY plane, and 90 is straight up (positive Z axis). If the altitude is -90 or 90, the azimuth is assumed to be 0.

Parameters

vectorA
[list] The input vector.

mantissa
[int, optional] The length of the desired mantissa. The default is 6.

Returns

dict
The dictionary containing the azimuth and altitude angles in degrees. The keys in the dictionary are 'azimuth' and 'altitude'.

`static Bisect`(*vectorA*, *vectorB*)

Compute the bisecting vector of two input vectors.

Parameters

vectorA
[list] The first input vector.

vectorB
[list] The second input vector.

Returns

dict
The bisecting vector.

`static ByAzimuthAltitude`(*azimuth: float*, *altitude: float*, *north: float = 0*, *reverse: bool = False*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the vector specified by the input azimuth and altitude angles.

Parameters

azimuth
[float] The input azimuth angle in degrees. The angle is computed in an anti-clockwise fashion. 0 is considered North, 90 East, 180 is South, 270 is West

altitude
[float] The input altitude angle in degrees from the XY plane. Positive is above the XY plane. Negative is below the XY plane

north
[float , optional] The angle of the north direction in degrees measured from positive Y-axis. The angle is added in anti-clockwise fashion. 0 is considered along the positive Y-axis, 90 is along the positive X-axis, 180 is along the negative Y-axis, and 270 along the negative Y-axis.

reverse
[bool , optional] If set to True the direction of the vector is computed from the end point towards the origin. Otherwise, it is computed from the origin towards the end point.

mantissa
[int , optional] The desired mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The resulting vector.

`static ByCoordinates`(*x*, *y*, *z*)

Creates a vector by the specified x, y, z inputs.

Parameters

x
[float] The X coordinate.

y
[float] The Y coordinate.

z
[float] The Z coodinate.

Returns

list
The created vector.

`static ByVertices`(*vertices*, *normalize: bool = True*, *mantissa: int = 6*)

Creates a vector by the specified input list of vertices.

Parameters

vertices
[list] The the input list of topologic vertices. The first element in the list is considered the start vertex. The last element in the list is considered the end vertex.

normalize
[bool , optional] If set to True, the resulting vector is normalized (i.e. its length is set to 1)

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The created vector.

`static CompassAngle`(*vectorA*, *vectorB*, *mantissa=6*, *tolerance=0.0001*)

Returns the horizontal compass angle in degrees between the two input vectors. The angle is measured in clockwise fashion. 0 is along the positive Y-axis, 90 is along the positive X axis. Only the first two elements in the input vectors are considered.

Parameters

vectorA
[list] The first vector.

vectorB
[list] The second vector.

mantissa
[int, optional] The length of the desired mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

float
The horizontal compass angle in degrees between the two input vectors.

`static Coordinates`(*vector*, *outputType='xyz'*, *mantissa: int = 6*)

Returns the coordinates of the input vector.

Parameters

vector
[list] The input vector.

outputType
[string, optional] The desired output type. Could be any permutation or substring of "xyz" or the string "matrix". The default is "xyz". The input is case insensitive and the coordinates will be returned in the specified order.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The coordinates of the input vertex.

`static Cross`(*vectorA*, *vectorB*, *mantissa=6*, *tolerance=0.0001*)

Returns the cross product of the two input vectors. The resulting vector is perpendicular to the plane defined by the two input vectors.

Parameters

vectorA
[list] The first vector.

vectorB
[list] The second vector.

mantissa
[int, optional] The length of the desired mantissa. The default is 6.

tolerance
[float, optional] the desired tolerance. The default is 0.0001.

Returns

list
The vector representing the cross product of the two input vectors.

`static Dot`(*vectorA*, *vectorB*, *mantissa=6*)

Returns the dot product of the two input vectors which is a measure of how much they are aligned.

Parameters

vectorA
[list] The first vector.

vectorB
[list] The second vector.

mantissa
[int, optional] The length of the desired mantissa. The default is 6.

Returns

list
The vector representing the cross product of the two input vectors.

`static Down()`

Returns the vector representing the *down* direction. In Topologic, the negative ZAxis direction is considered *down* ([0, 0, -1]).

Returns

list
The vector representing the *down* direction.

`static East()`

Returns the vector representing the *east* direction. In Topologic, the positive XAxis direction is considered *east* ([1, 0, 0]).

Returns

list
The vector representing the *east* direction.

`static IsAntiParallel(vectorA, vectorB)`

Returns True if the input vectors are anti-parallel. Returns False otherwise.

Parameters

vectorA
[list] The first input vector.

vectorB
[list] The second input vector.

Returns

bool
True if the input vectors are anti-parallel. False otherwise.

`static IsCollinear(vectorA, vectorB)`

Returns True if the input vectors are collinear (parallel or anti-parallel). Returns False otherwise.

Parameters

vectorA
[list] The first input vector.

vectorB
[list] The second input vector.

Returns

bool
True if the input vectors are collinear (parallel or anti-parallel). False otherwise.

`static IsParallel(vectorA, vectorB)`

Returns True if the input vectors are parallel. Returns False otherwise.

Parameters

vectorA
[list] The first input vector.

vectorB
[list] The second input vector.

Returns

bool
True if the input vectors are parallel. False otherwise.

`static IsSame`(*vectorA*, *vectorB*, *tolerance=0.0001*)

Returns True if the input vectors are the same. Returns False otherwise.

Parameters

vectorA
[list] The first input vector.

vectorB
[list] The second input vector.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the input vectors are the same. False otherwise.

`static Length`(*vector*, *mantissa: int = 6*)

Returns the length of the input vector.

Parameters

vector
[list] The input vector.

mantissa
[int] The length of the desired mantissa. The default is 6.

Returns

float
The length of the input vector.

`static Magnitude`(*vector*, *mantissa: int = 6*)

Returns the magnitude of the input vector.

Parameters

vector
[list] The input vector.

mantissa
[int] The length of the desired mantissa. The default is 6.

Returns

float
The magnitude of the input vector.

`static Multiply`(*vector*, *magnitude*, *tolerance=0.0001*)

Multiplies the input vector by the input magnitude.

Parameters

vector
[list] The input vector.

magnitude
[float] The input magnitude.

tolerance
[float, optional] the desired tolerance. The default is 0.0001.

Returns

list
The created vector that multiplies the input vector by the input magnitude.

`static Normalize(vector)`

Returns a normalized vector of the input vector. A normalized vector has the same direction as the input vector, but its magnitude is 1.

Parameters

vector
[list] The input vector.

Returns

list
The normalized vector.

`static North()`

Returns the vector representing the *north* direction. In Topologic, the positive YAxis direction is considered *north* ([0, 1, 0]).

Returns

list
The vector representing the *north* direction.

`static NorthEast()`

Returns the vector representing the *northeast* direction. In Topologic, the positive YAxis direction is considered *north* and the positive XAxis direction is considered *east*. Therefore *northeast* is ([1, 1, 0]).

Returns

list
The vector representing the *northeast* direction.

`static NorthWest()`

Returns the vector representing the *northwest* direction. In Topologic, the positive YAxis direction is considered *north* and the negative XAxis direction is considered *west*. Therefore *northwest* is ([-1, 1, 0]).

Returns

list
The vector representing the *northwest* direction.

`static Reverse(vector)`

Returns a reverse vector of the input vector. A reverse vector multiplies all components by -1.

Parameters

vector
[list] The input vector.

Returns

list
The normalized vector.

`static SetMagnitude`(*vector: list*, *magnitude: float*) → list

Sets the magnitude of the input vector to the input magnitude.

Parameters

vector
[list] The input vector.

magnitude
[float] The desired magnitude.

Returns

list
The created vector.

`static South`()

Returns the vector representing the *south* direction. In Topologic, the negative YAxis direction is considered *south* ([0, -1, 0]).

Returns

list
The vector representing the *south* direction.

`static SouthEast`()

Returns the vector representing the *southeast* direction. In Topologic, the negative YAxis direction is considered *south* and the positive XAxis direction is considered *east*. Therefore *southeast* is ([1, -1, 0]).

Returns

list
The vector representing the *southeast* direction.

`static SouthWest`()

Returns the vector representing the *southwest* direction. In Topologic, the negative YAxis direction is considered *south* and the negative XAxis direction is considered *west*. Therefore *southwest* is ([-1, -1, 0]).

Returns

list
The vector representing the *southwest* direction.

`static Subtract`(*vectorA*, *vectorB*)

Subtracts the second input vector from the first input vector.

Parameters

vectorA
[list] The first vector.

vectorB
[list] The second vector.

Returns

list
The vector resulting from subtracting the second input vector from the first input vector.

`static Sum`(*vectors: list*)

Returns the sum vector of the input vectors.

Parameters

vectors
[list] The input list of vectors.

Returns

list
The sum vector of the input list of vectors.

`static TransformationMatrix(vectorA, vectorB)`

Returns the transformation matrix needed to align vectorA with vectorB.

Parameters

vectorA
[list] The input vector to be transformed.

vectorB
[list] The desired vector with which to align vectorA.

Returns

list
Transformation matrix that follows the Blender software convention (nested list)

`static Up()`

Returns the vector representing the up direction. In Topologic, the positive ZAxis direction is considered "up" ([0, 0, 1]).

Returns

list
The vector representing the "up" direction.

`static West()`

Returns the vector representing the *west* direction. In Topologic, the negative XAxis direction is considered *west* ([-1, 0, 0]).

Returns

list
The vector representing the *west* direction.

`static XAxis()`

Returns the vector representing the XAxis ([1, 0, 0])

Returns

list
The vector representing the XAxis.

`static YAxis()`

Returns the vector representing the YAxis ([0, 1, 0])

Returns

list
The vector representing the YAxis.

`static ZAxis()`

Returns the vector representing the ZAxis ([0, 0, 1])

Returns

list
The vector representing the ZAxis.

topologicpy.Vertex module

class topologicpy.Vertex.**Vertex**
Bases: object

Methods

AreCollinear(vertices[, mantissa, tolerance])	Returns True if the input list of vertices form a straight line.
AreIpsilateral(vertices, face)	Returns True if the input list of vertices are on one side of a face.
AreIpsilateralCluster(cluster, face)	Returns True if the two input vertices are on the same side of the input face.
AreOnSameSide(vertices, face)	Returns True if the two input vertices are on the same side of the input face.
AreOnSameSideCluster(cluster, face)	Returns True if the two input vertices are on the same side of the input face.
ByCoordinates(*args, **kwargs)	Creates a vertex at the coordinates specified by the x, y, z inputs.
Centroid(vertices[, mantissa])	Returns the centroid of the input list of vertices.
Clockwise2D(vertices)	Sorts the input list of vertices in a clockwise fashion.
Coordinates(vertex[, outputType, mantissa])	Returns the coordinates of the input vertex.
CounterClockwise2D(vertices[, mantissa])	Sorts the input list of vertices in a counterclockwise fashion.
Degree(vertex, hostTopology[, topologyType])	Returns the vertex degree (the number of super topologies connected to it).
Distance(vertex, topology[, ...])	Returns the distance between the input vertex and the input topology.
EnclosingCell(vertex, topology[, exclusive, ...])	Returns the list of Cells found in the input topology that enclose the input vertex.
ExternalBoundary(vertex)	Returns the external boundary (self) of the input vertex.
Fuse(vertices[, mantissa, tolerance])	Returns a list of vertices where vertices within a specified tolerance distance are fused while retaining duplicates, ensuring that vertices with nearly identical coordinates are replaced by a single shared coordinate.
IncomingEdges(vertex, hostTopology[, tolerance])	Returns the incoming edges connected to a vertex.
Index(vertex, vertices[, strict, tolerance])	Returns index of the input vertex in the input list of vertices
InterpolateValue(vertex, vertices[, n, key, ...])	Interpolates the value of the input vertex based on the values of the *n* nearest vertices.
IsCoincident(vertexA, vertexB[, tolerance, ...])	Returns True if the input vertexA is coincident with the input vertexB.
IsExternal(vertex, topology[, tolerance, silent])	Returns True if the input vertex is external to the input topology.

continues on next page

Table 7 – continued from previous page

IsInternal(vertex, topology[, tolerance, silent])	Returns True if the input vertex is inside the input topology.
IsPeripheral(vertex, topology[, tolerance, ...])	Returns True if the input vertex is peripheral to the input topology.
NearestVertex(vertex, topology[, useKDTree, ...])	Returns the vertex found in the input topology that is the nearest to the input vertex.
Origin()	Returns a vertex with coordinates (0, 0, 0)
OutgoingEdges(vertex, hostTopology[, tolerance])	Returns the outgoing edges connected to a vertex.
PerpendicularDistance(vertex, face[, mantissa])	Returns the perpendicular distance between the input vertex and the input face.
PlaneEquation(vertices[, mantissa])	Returns the equation of the average plane passing through a list of vertices.
Point([x, y, z])	Creates a point (vertex) using the input parameters
Project(vertex, face[, direction, mantissa])	Returns a vertex that is the projection of the input vertex unto the input face.
X(vertex[, mantissa])	Returns the X coordinate of the input vertex.
Y(vertex[, mantissa])	Returns the Y coordinate of the input vertex.
Z(vertex[, mantissa])	Returns the Z coordinate of the input vertex.

static AreCollinear(*vertices: list*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns True if the input list of vertices form a straight line. Returns False otherwise.

Parameters

vertices
[list] The input list of vertices.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the input vertices are on the same side of the face. False otherwise.

static AreIpsilateral(*vertices: list*, *face*) → bool

Returns True if the input list of vertices are on one side of a face. Returns False otherwise. If at least one of the vertices is on the face, this method return True.

Parameters

vertices
[list] The input list of vertices.

face
[topologic_core.Face] The input face

Returns

bool
True if the input vertices are on the same side of the face. False otherwise. If at least one of the vertices is on the face, this method return True.

`static AreIpsilateralCluster`(*cluster*, *face*) → bool

Returns True if the two input vertices are on the same side of the input face. Returns False otherwise. If at least one of the vertices is on the face, this method return True.

Parameters

cluster
[topologic_core.Cluster] The input list of vertices.

face
[topologic_core.Face] The input face

Returns

bool
True if the input vertices are on the same side of the face. False otherwise. If at least one of the vertices is on the face, this method return True.

`static AreOnSameSide`(*vertices: list*, *face*) → bool

Returns True if the two input vertices are on the same side of the input face. Returns False otherwise. If at least one of the vertices is on the face, this method return True.

Parameters

vertices
[list] The input list of vertices.

face
[topologic_core.Face] The input face

Returns

bool
True if the input vertices are on the same side of the face. False otherwise. If at least one of the vertices is on the face, this method return True.

`static AreOnSameSideCluster`(*cluster*, *face*) → bool

Returns True if the two input vertices are on the same side of the input face. Returns False otherwise. If at least one of the vertices is on the face, this method return True.

Parameters

cluster
[topologic_core.Cluster] The input list of vertices.

face
[topologic_core.Face] The input face

Returns

bool
True if the input vertices are on the same side of the face. False otherwise. If at least one of the vertices is on the face, this method return True.

`static ByCoordinates`(**args*, ***kwargs*)

Creates a vertex at the coordinates specified by the x, y, z inputs. You can call this method using a list of coordinates or individually. Examples: v = Vertex.ByCoordinates(3.4, 5.7, 2.8) v = Vertex.ByCoordinates([3.4, 5.7, 2.8]) v = Vertex.ByCoordinates(x=3.4, y=5.7, z=2.8)

Parameters

x
[float , optional] The X coordinate. The default is 0.

y
[float , optional] The Y coordinate. The default is 0.

z
[float , optional] The Z coordinate. The defaults is 0.

Returns

topologic_core.Vertex
The created vertex.

`static Centroid`(*vertices: list*, *mantissa: int = 6*)

Returns the centroid of the input list of vertices.

Parameters

vertices
[list] The input list of vertices

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic_core.Vertex
The computed centroid of the input list of vertices

`static Clockwise2D`(*vertices*)

Sorts the input list of vertices in a clockwise fashion. This method assumes that the vertices are on the XY plane. The Z coordinate is ignored.

Parameters

vertices
[list] The input list of vertices

Returns

list
The input list of vertices sorted in a counter clockwise fashion

`static Coordinates`(*vertex*, *outputType: str = 'xyz'*, *mantissa: int = 6*) → list

Returns the coordinates of the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

outputType
[string, optional] The desired output type. Could be any permutation or substring of "xyz" or the string "matrix". The default is "xyz". The input is case insensitive and the coordinates will be returned in the specified order.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The coordinates of the input vertex.

`static CounterClockwise2D`(*vertices: list*, *mantissa: int = 6*)

Sorts the input list of vertices in a counterclockwise fashion. This method assumes that the vertices are on the XY plane. The Z coordinate is ignored.

Parameters

vertices
[list] The input list of vertices

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The input list of vertices sorted in a counter clockwise fashion

`static Degree`(*vertex*, *hostTopology*, *topologyType: str = 'edge'*)

Returns the vertex degree (the number of super topologies connected to it). See https://en.wikipedia.org/wiki/Degree_(graph_theory).

Parameters

vertex
[topologic_core.Vertex] The input vertex.

hostTopology
[topologic_core.Topology] The input host topology in which to search for the connected super topologies.

topologyType
[str , optional] The topology type to search for. This can be any of "edge", "wire", "face", "shell", "cell", "cellcomplex", "cluster". It is case insensitive. If set to None, the immediate supertopology type is searched for. The default is None.

Returns

int
The number of super topologies connected to this vertex

`static Distance`(*vertex*, *topology*, *includeCentroid: bool = True*, *mantissa: int = 6*) → float

Returns the distance between the input vertex and the input topology. This method returns the distance to the closest sub-topology in the input topology, optionally including its centroid.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

topology
[topologic_core.Topology] The input topology.

includeCentroid
[bool] If set to True, the centroid of the input topology will be considered in finding the nearest subTopology to the input vertex. The default is True.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The distance between the input vertex and the input topology.

static EnclosingCell(*vertex*, *topology*, *exclusive: bool = True*, *mantissa: int = 6*, *tolerance: float = 0.0001*) → list

Returns the list of Cells found in the input topology that enclose the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

topology
[topologic_core.Topology] The input topology.

exclusive
[bool , optional] If set to True, return only the first found enclosing cell. The default is True.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The tolerance for computing if the input vertex is enclosed in a cell. The default is 0.0001.

Returns

list
The list of enclosing cells.

static ExternalBoundary(*vertex*)

Returns the external boundary (self) of the input vertex. This method is trivial, but included for completeness.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

Returns

topologic_core.Vertex
The external boundary of the input vertex. This is the input vertex itself.

static Fuse(*vertices: list*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a list of vertices where vertices within a specified tolerance distance are fused while retaining duplicates, ensuring that vertices with nearly identical coordinates are replaced by a single shared coordinate.

Parameters

vertices
[list] The input list of topologic vertices.

mantissa
[int , optional] The desired length of the mantissa for retrieving vertex coordinates. The default is 6.

tolerance
[float , optional] The desired tolerance for computing if vertices need to be fused. Any vertices that are closer to each other than this tolerance will be fused. The default is 0.0001.

Returns

list
The list of fused vertices. This list contains the same number of vertices and in the same

order as the input list of vertices. However, the coordinates of these vertices have now been modified so that they are exactly the same with other vertices that are within the tolerance distance.

`static IncomingEdges`(*vertex*, *hostTopology*, *tolerance: float = 0.0001*) → list

Returns the incoming edges connected to a vertex. An edge is considered incoming if its end vertex is coincident with the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

hostTopology
[topologic_core.Topology] The input host topology to which the vertex belongs.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of incoming edges

`static Index`(*vertex*, *vertices: list*, *strict: bool = False*, *tolerance: float = 0.0001*) → int

Returns index of the input vertex in the input list of vertices

Parameters

vertex
[topologic_core.Vertex] The input vertex.

vertices
[list] The input list of vertices.

strict
[bool , optional] If set to True, the vertex must be strictly identical to the one found in the list. Otherwise, a distance comparison is used. The default is False.

tolerance
[float , optional] The tolerance for computing if the input vertex is identical to a vertex from the list. The default is 0.0001.

Returns

int
The index of the input vertex in the input list of vertices.

`static InterpolateValue`(*vertex*, *vertices: list*, *n: int = 3*, *key: str = 'intensity'*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Interpolates the value of the input vertex based on the values of the *n* nearest vertices.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

vertices
[list] The input list of vertices.

n
[int , optional] The maximum number of nearest vertices to consider. The default is 3.

key
[str , optional] The key that holds the value to be interpolated in the dictionaries of the vertices. The default is "intensity".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The tolerance for computing if the input vertex is coincident with another vertex in the input list of vertices. The default is 0.0001.

Returns

topologic_core.vertex
The input vertex with the interpolated value stored in its dictionary at the key specified by the input key. Other keys and values in the dictionary are preserved.

`static IsCoincident`(*vertexA*, *vertexB*, *tolerance: float = 0.0001*, *silent: bool = False*) → bool

Returns True if the input vertexA is coincident with the input vertexB. Returns False otherwise.

Parameters

vertexA
[topologic_core.Vertex] The first input vertex.

vertexB
[topologic_core.Vertex] The second input vertex.

tolerance
[float , optional] The tolerance for computing if the input vertexA is coincident with the input vertexB. The default is 0.0001.

Returns

bool
True if the input vertexA is coincident with the input vertexB. False otherwise.

`static IsExternal`(*vertex*, *topology*, *tolerance: float = 0.0001*, *silent: bool = False*) → bool

Returns True if the input vertex is external to the input topology. Returns False otherwise.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

topology
[topologic_core.Topology] The input topology.

tolerance
[float , optional] The tolerance for computing if the input vertex is external to the input topology. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

bool
True if the input vertex is external to the input topology. False otherwise.

`static IsInternal`(*vertex*, *topology*, *tolerance: float = 0.0001*, *silent: bool = False*) → bool

Returns True if the input vertex is inside the input topology. Returns False otherwise.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

topology
[topologic_core.Topology] The input topology.

tolerance
[float , optional] The tolerance for computing if the input vertex is internal to the input topology. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

bool
True if the input vertex is internal to the input topology. False otherwise.

`static IsPeripheral`(*vertex*, *topology*, *tolerance: float = 0.0001*, *silent: bool = False*) → bool

Returns True if the input vertex is peripheral to the input topology. Returns False otherwise. A vertex is said to be peripheral to the input topology if: 01. Vertex: If it is internal to it (i.e. coincident with it). 02. Edge: If it is internal to its start or end vertices. 03. Manifold open wire: If it is internal to its start or end vertices. 04. Manifold closed wire: If it is internal to any of its vertices. 05. Non-manifold wire: If it is internal to any of its vertices that has a vertex degree of 1. 06. Face: If it is internal to any of its edges or vertices. 07. Shell: If it is internal to external boundary 08. Cell: If it is internal to any of its faces, edges, or vertices. 09. CellComplex: If it is peripheral to its external boundary. 10. Cluster: If it is peripheral to any of its free topologies. (See Cluster.FreeTopologies)

Parameters

vertex
[topologic_core.Vertex] The input vertex.

topology
[topologic_core.Topology] The input topology.

tolerance
[float , optional] The tolerance for computing if the input vertex is peripheral to the input topology. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

bool
True if the input vertex is peripheral to the input topology. False otherwise.

`static NearestVertex`(*vertex*, *topology*, *useKDTree: bool = True*, *mantissa: int = 6*)

Returns the vertex found in the input topology that is the nearest to the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

topology
[topologic_core.Topology] The input topology to be searched for the nearest vertex.

useKDTree
[bool , optional] if set to True, the algorithm will use a KDTree method to search for the nearest vertex. The default is True.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic_core.Vertex
The nearest vertex.

`static Origin()`

Returns a vertex with coordinates (0, 0, 0)

Parameters

Returns

topologic_core.Vertex

`static OutgoingEdges(`*vertex*, *hostTopology*, *tolerance: float = 0.0001*`)` → list

Returns the outgoing edges connected to a vertex. An edge is considered incoming if its start vertex is coincident with the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

hostTopology
[topologic_core.Topology] The input host topology to which the vertex belongs.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of outgoing edges

`static PerpendicularDistance(`*vertex*, *face*, *mantissa: int = 6*`)`

Returns the perpendicular distance between the input vertex and the input face. The face is considered to be infinite.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

face
[topologic_core.Face] The input face.

mantissa: int , optional
The desired length of the mantissa. The default is 6.

Returns

float
The distance between the input vertex and the input topology.

static PlaneEquation(*vertices*, *mantissa: int = 6*)

Returns the equation of the average plane passing through a list of vertices.

Parameters

vertices
[list] The input list of vertices

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

dict
The dictionary containing the values of a, b, c, d for the plane equation in the form of ax+by+cz+d=0. The keys in the dictionary are [“a”, “b”, “c”. “d”]

static Point(*x=0*, *y=0*, *z=0*)

Creates a point (vertex) using the input parameters

Parameters

x
[float , optional.] The desired x coordinate. The default is 0.

y
[float , optional.] The desired y coordinate. The default is 0.

z
[float , optional.] The desired z coordinate. The default is 0.

Returns

topologic_core.Vertex

static Project(*vertex*, *face*, *direction: bool = None*, *mantissa: int = 6*)

Returns a vertex that is the projection of the input vertex unto the input face.

Parameters

vertex
[topologic_core.Vertex] The input vertex to project unto the input face.

face
[topologic_core.Face] The input face that receives the projection of the input vertex.

direction
[vector, optional] The direction in which to project the input vertex unto the input face. If not specified, the direction of the projection is the normal of the input face. The default is None.

mantissa
[int , optional] The length of the desired mantissa. The default is 6.

Returns

topologic_core.Vertex
The projected vertex.

static X(*vertex*, *mantissa: int = 6*) → float

Returns the X coordinate of the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The X coordinate of the input vertex.

`static Y(vertex, mantissa: int = 6) → float`

Returns the Y coordinate of the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The Y coordinate of the input vertex.

`static Z(vertex, mantissa: int = 6) → float`

Returns the Z coordinate of the input vertex.

Parameters

vertex
[topologic_core.Vertex] The input vertex.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The Z coordinate of the input vertex.

topologicpy.Wire module

`class topologicpy.Wire.Wire`

Bases: `object`

Methods

Arc(startVertex, middleVertex, endVertex[, ...])	Creates an arc.
ArcByEdge([sagitta, absolute, sides, close, ...])	Creates an arc.
BoundingRectangle(topology[, optimize, ...])	Returns a wire representing a bounding rectangle of the input topology.
ByEdges(edges[, orient, tolerance])	Creates a wire from the input list of edges.
ByEdgesCluster(cluster[, tolerance])	Creates a wire from the input cluster of edges.
ByOffset(wire[, offset, offsetKey, ...])	Creates an offset wire from the input wire.

continues on next page

Table 8 – continued from previous page

ByOffsetArea(wire, area[, offsetKey, ...])	Creates an offset wire from the input wire based on the input area.
ByVertices(vertices[, close, tolerance])	Creates a wire from the input list of vertices.
ByVerticesCluster(cluster[, close])	Creates a wire from the input cluster of vertices.
Circle([origin, radius, sides, fromAngle, ...])	Creates a circle.
Close(wire[, mantissa, tolerance])	Closes the input wire
ConcaveHull(topology[, k, mantissa, tolerance])	Returns a wire representing the 2D concave hull of the input topology.
ConvexHull(topology[, mantissa, tolerance])	Returns a wire representing the 2D convex hull of the input topology.
Cycles(wire[, maxVertices, tolerance])	Returns the closed circuits of wires found within the input wire.
Edges(wire)	Returns the edges of the input wire.
Einstein([origin, radius, direction, ...])	Creates an aperiodic monotile, also called an 'einstein' tile (meaning one tile in German, not the name of the famous physicist).
Ellipse([origin, inputMode, width, length, ...])	Creates an ellipse and returns all its geometry and parameters.
EllipseAll([origin, inputMode, width, ...])	Creates an ellipse and returns all its geometry and parameters.
EndVertex(wire)	Returns the end vertex of the input wire.
ExteriorAngles(wire[, tolerance, mantissa])	Returns the exterior angles of the input wire in degrees.
ExternalBoundary(wire)	Returns the external boundary (cluster of vertices where degree == 1) of the input wire.
Fillet(wire[, radius, sides, radiusKey, ...])	Fillets (rounds) the interior and exterior corners of the input wire given the input radius.
InteriorAngles(wire[, tolerance, mantissa])	Returns the interior angles of the input wire in degrees.
Interpolate(wires[, n, outputType, mapping, ...])	Creates *n* number of wires that interpolate between wireA and wireB.
Invert(wire)	Creates a wire that is an inverse (mirror) of the input wire.
IsClosed(wire)	Returns True if the input wire is closed.
IsManifold(wire[, silent])	Returns True if the input wire is manifold.
IsSimilar(wireA, wireB[, angTolerance, ...])	Returns True if the input wires are similar.
Length(wire[, mantissa])	Returns the length of the input wire.
Line([origin, length, direction, sides, ...])	Creates a straight line wire using the input parameters.
Miter(wire[, offset, offsetKey, tolerance, ...])	Fillets (rounds) the interior and exterior corners of the input wire given the input radius.
Normal(wire[, outputType, mantissa])	Returns the normal vector to the input wire.
OrientEdges(wire, vertexA[, ...])	Returns a correctly oriented head-to-tail version of the input wire.
Planarize(wire[, origin, mantissa, tolerance])	Returns a planarized version of the input wire.
Project(wire, face[, direction, mantissa, ...])	Creates a projection of the input wire unto the input face.
Rectangle([origin, width, length, ...])	Creates a rectangle.
RemoveCollinearEdges(wire[, angTolerance, ...])	Removes any collinear edges in the input wire.
Reverse(wire[, transferDictionaries, tolerance])	Creates a wire that has the reverse direction of the input wire.

continues on next page

Table 8 – continued from previous page

Roof(face[, angle, boundary, tolerance])	Creates a hipped roof through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com>
Simplify(wire[, method, tolerance, silent])	Simplifies the input wire edges based on the selected algorithm: Douglas-Peucker or Visvalingam–Whyatt.
Skeleton(face[, boundary, tolerance])	Creates a straight skeleton.
Spiral([origin, radiusA, radiusB, height, ...])	Creates a spiral.
Split(wire)	Splits the input wire into segments at its intersections (i.e.
Square([origin, size, direction, placement, ...])	Creates a square.
Squircle([origin, radius, sides, a, b, ...])	Creates a Squircle which is a hybrid between a circle and a square.
Star([origin, radiusA, radiusB, rays, ...])	Creates a star.
StartEndVertices(wire)	Returns the start and end vertices of the input wire.
StartVertex(wire)	Returns the start vertex of the input wire.
Trapezoid([origin, widthA, widthB, offsetA, ...])	Creates a trapezoid.
VertexByDistance(wire[, distance, origin, ...])	Creates a vertex along the input wire offset by the input distance from the input origin.
VertexByParameter(wire[, u])	Creates a vertex along the input wire offset by the input *u* parameter.
VertexDistance(wire, vertex[, origin, ...])	Returns the distance, computed along the input wire of the input vertex from the input origin vertex.
Vertices(wire)	Returns the list of vertices of the input wire.

static Arc(*startVertex*, *middleVertex*, *endVertex*, *sides: int = 16*, *close: bool = True*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates an arc. The base chord will be parallel to the x-axis and the height will point in the positive y-axis direction.

Parameters

startVertex
[topologic_core.Vertex] The start vertex of the arc.

middleVertex
[topologic_core.Vertex] The middle vertex (apex) of the arc.

endVertex
[topologic_core.Vertex] The end vertex of the arc.

sides
[int , optional] The number of sides of the arc. The default is 16.

close
[bool , optional] If set to True, the arc will be closed by connecting the last vertex to the first vertex. Otherwise, it will be left open.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The created arc.

`ArcByEdge`(*sagitta: float = 1*, *absolute: bool = True*, *sides: int = 16*, *close: bool = True*, *tolerance: float = 0.0001*, *silent: bool = False*)

Creates an arc. The base chord will be parallel to the x-axis and the height will point in the positive y-axis direction.

Parameters

edge
[topologic_core.Edge] The location of the start vertex of the arc.

sagitta
[float , optional] The length of the sagitta. In mathematics, the sagitta is the line connecting the center of a chord to the apex (or highest point) of the arc subtended by that chord. The default is 1.

absolute
[bool , optional] If set to True, the sagitta length is treated as an absolute value. Otherwise, it is treated as a ratio based on the length of the edge. For example, if the length of the edge is 10, the sagitta is set to 0.5, and absolute is set to False, the sagitta length will be 5. The default is True.

sides
[int , optional] The number of sides of the arc. The default is 16.

close
[bool , optional] If set to True, the arc will be closed by connecting the last vertex to the first vertex. Otherwise, it will be left open.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The created arc.

`static BoundingRectangle`(*topology*, *optimize: int = 0*, *mantissa: int = 6*, *tolerance=0.0001*)

Returns a wire representing a bounding rectangle of the input topology. The returned wire contains a dictionary with key "zrot" that represents rotations around the Z axis. If applied the resulting wire will become axis-aligned.

Parameters

topology
[topologic_core.Topology] The input topology.

optimize
[int , optional] If set to an integer from 1 (low optimization) to 10 (high optimization), the method will attempt to optimize the bounding rectangle so that it reduces its surface area. The minimum optimization number of 0 will result in an axis-aligned bounding rectangle. A maximum optimization number of 10 will attempt to reduce the bounding rectangle's area by 50%. The default is 0.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The bounding rectangle of the input topology.

`static ByEdges`(*edges: list*, *orient: bool = False*, *tolerance: float = 0.0001*)

Creates a wire from the input list of edges.

Parameters

edges
[list] The input list of edges.

orient
[bool , optional] If set to True the edges are oriented head to tail. Otherwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001

Returns

topologic_core.Wire
The created wire.

`static ByEdgesCluster`(*cluster*, *tolerance: float = 0.0001*)

Creates a wire from the input cluster of edges.

Parameters

cluster
[topologic_core.Cluster] The input cluster of edges.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created wire.

`static ByOffset`(*wire*, *offset: float = 1.0*, *offsetKey: str = 'offset'*, *stepOffsetA: float = 0*, *stepOffsetB: float = 0*, *stepOffsetKeyA: str = 'stepOffsetA'*, *stepOffsetKeyB: str = 'stepOffsetB'*, *reverse: bool = False*, *bisectors: bool = False*, *transferDictionaries: bool = False*, *epsilon: float = 0.01*, *tolerance: float = 0.0001*, *silent: bool = False*, *numWorkers: int = None*)

Creates an offset wire from the input wire. A positive offset value results in an offset to the interior of an anti-clockwise wire.

Parameters

wire
[topologic_core.Wire] The input wire.

offset
[float , optional] The desired offset distance. The default is 1.0.

offsetKey
[str , optional] The edge dictionary key under which to find the offset value. If a value cannot be found, the offset input parameter value is used instead. The default is "offset".

stepOffsetA
[float , optional] The amount to offset along the previous edge when transitioning between parallel edges with different offsets. The default is 0.

stepOffsetB
[float , optional] The amount to offset along the next edge when transitioning between parallel edges with different offsets. The default is 0.

stepOffsetKeyA
[str , optional] The vertex dictionary key under which to find the step offset A value. If a value cannot be found, the stepOffsetA input parameter value is used instead. The default is "stepOffsetA".

stepOffsetKeyB
[str , optional] The vertex dictionary key under which to find the step offset B value. If a value cannot be found, the stepOffsetB input parameter value is used instead. The default is "stepOffsetB".

reverse
[bool , optional] If set to True, the direction of offsets is reversed. Otherwise, it is not. The default is False.

bisectors
[bool , optional] If set to True, The bisectors (seams) edges will be included in the returned wire. The default is False.

transferDictionaries
[bool , optional] If set to True, the dictionaries of the original wire, its edges, and its vertices are transfered to the new wire. Otherwise, they are not. The default is False.

epsilon
[float , optional] The desired epsilon (another form of tolerance for shortest edge to remove). The default is 0.01. (This is set to a larger number as it was found to work better)

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

numWorkers
[int , optional] Number of workers run in parallel to process. If you set it to 1, no parallel processing will take place. The default is None which causes the algorithm to use twice the number of cpu cores in the host computer.

Returns

topologic_core.Wire
The created wire.

`static ByOffsetArea`(*wire*, *area*, *offsetKey='offset'*, *minOffsetKey='minOffset'*, *maxOffsetKey='maxOffset'*, *defaultMinOffset=0*, *defaultMaxOffset=1*, *maxIterations=1*, *tolerance=0.0001*, *silent=False*, *numWorkers=None*)

Creates an offset wire from the input wire based on the input area.

Parameters

wire
[topologic_core.Wire] The input wire.

area
[float] The desired area of the created wire.

offsetKey
[str , optional] The edge dictionary key under which to store the offset value. The default is "offset".

minOffsetKey
[str , optional] The edge dictionary key under which to find the desired minimum edge offset value. If a value cannot be found, the defaultMinOffset input parameter value is used instead. The default is "minOffset".

maxOffsetKey
[str , optional] The edge dictionary key under which to find the desired maximum edge offset value. If a value cannot be found, the defaultMaxOffset input parameter value is used instead. The default is "maxOffset".

defaultMinOffset
[float , optional] The desired minimum edge offset distance. The default is 0.

defaultMaxOffset
[float , optional] The desired maximum edge offset distance. The default is 1.

maxIterations: int , optional
The desired maximum number of iterations to attempt to converge on a solution. The default is 1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

numWorkers
[int , optional] Number of workers run in parallel to process. If you set it to 1, no parallel processing will take place. The default is None which causes the algorithm to use twice the number of cpu cores in the host computer.

Returns

topologic_core.Wire
The created wire.

`static ByVertices`(*vertices: list*, *close: bool = True*, *tolerance: float = 0.0001*)

Creates a wire from the input list of vertices.

Parameters

vertices
[list] the input list of vertices.

close
[bool , optional] If True the last vertex will be connected to the first vertex to close the wire. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created wire.

`static ByVerticesCluster`(*cluster*, *close: bool = True*)

Creates a wire from the input cluster of vertices.

Parameters

cluster
[topologic_core.cluster] the input cluster of vertices.

close
[bool , optional] If True the last vertex will be connected to the first vertex to close the wire. The default is True.

Returns

topologic_core.Wire
The created wire.

`static Circle`(*origin=None*, *radius: float = 0.5*, *sides: int = 16*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *close: bool = True*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a circle.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the circle. The default is None which results in the circle being placed at (0, 0, 0).

radius
[float , optional] The radius of the circle. The default is 0.5.

sides
[int , optional] The desired number of sides of the circle. The default is 16.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the circle. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the circle. The default is 360.

close
[bool , optional] If set to True, arcs will be closed by connecting the last vertex to the first vertex. Otherwise, they will be left open.

direction
[list , optional] The vector representing the up direction of the circle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the circle. This can be “center”, “lowerleft”, “upperleft”, “lowerright”, or “upperright”. It is case insensitive. The default is “center”.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created circle.

`static Close`(*wire*, *mantissa=6*, *tolerance=0.0001*)

Closes the input wire

Parameters

wire
[topologic_core.Wire] The input wire.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The closed version of the input wire.

`static ConcaveHull`(*topology*, *k: int = 3*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a wire representing the 2D concave hull of the input topology. The vertices of the topology are assumed to be coplanar. Code based on Moreira, A and Santos, M Y, "CONCAVE HULL: A K-NEAREST NEIGHBOURS APPROACH FOR THE COMPUTATION OF THE REGION OCCUPIED BY A SET OF POINTS" GRAPP 2007 - International Conference on Computer Graphics Theory and Applications.

Parameters

topology
[topologic_core.Topology] The input topology.

k
[int, optional] The number of nearest neighbors to consider for each point when building the hull. Must be at least 3 for the algorithm to function correctly. Increasing *k* will produce a smoother, less concave hull, while decreasing *k* may yield a more detailed, concave shape. The default is 3.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The concave hull of the input topology.

`static ConvexHull`(*topology*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a wire representing the 2D convex hull of the input topology. The vertices of the topology are assumed to be coplanar.

Parameters

topology
[topologic_core.Topology] The input topology.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The convex hull of the input topology.

`static Cycles`(*wire*, *maxVertices: int = 4*, *tolerance: float = 0.0001*) → list

Returns the closed circuits of wires found within the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

maxVertices
[int , optional] The maximum number of vertices of the circuits to be searched. The default is 4.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

list
The list of circuits (closed wires) found within the input wire.

`static Edges`(*wire*) → list

Returns the edges of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

Returns

list
The list of edges.

`static Einstein`(*origin=None*, *radius: float = 0.5*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *mantissa: int = 6*)

Creates an aperiodic monotile, also called an 'einstein' tile (meaning one tile in German, not the name of the famous physicist). See https://arxiv.org/abs/2303.10798

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the tile. The default is None which results in the tiles first vertex being placed at (0, 0, 0).

radius
[float , optional] The radius of the hexagon determining the size of the tile. The default is 0.5.

direction
[list , optional] The vector representing the up direction of the ellipse. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the hexagon determining

the location of the tile. This can be “center”, or “lowerleft”. It is case insensitive. The default is “center”.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns
——-

topologic_core.Wire
The created wire.

`static Ellipse`(*origin=None*, *inputMode: int = 1*, *width: float = 2.0*, *length: float = 1.0*, *focalLength: float = 0.866025*, *eccentricity: float = 0.866025*, *majorAxisLength: float = 1.0*, *minorAxisLength: float = 0.5*, *sides: float = 32*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *close: bool = True*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an ellipse and returns all its geometry and parameters.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the ellipse. The default is None which results in the ellipse being placed at (0, 0, 0).

inputMode
[int , optional] The method by which the ellipse is defined. The default is 1. Based on the inputMode value, only the following inputs will be considered. The options are: 1. Width and Length (considered inputs: width, length) 2. Focal Length and Eccentricity (considered inputs: focalLength, eccentricity) 3. Focal Length and Minor Axis Length (considered inputs: focalLength, minorAxisLength) 4. Major Axis Length and Minor Axis Length (considered input: majorAxisLength, minorAxisLength)

width
[float , optional] The width of the ellipse. The default is 2.0. This is considered if the inputMode is 1.

length
[float , optional] The length of the ellipse. The default is 1.0. This is considered if the inputMode is 1.

focalLength
[float , optional] The focal length of the ellipse. The default is 0.866025. This is considered if the inputMode is 2 or 3.

eccentricity
[float , optional] The eccentricity of the ellipse. The default is 0.866025. This is considered if the inputMode is 2.

majorAxisLength
[float , optional] The length of the major axis of the ellipse. The default is 1.0. This is considered if the inputMode is 4.

minorAxisLength
[float , optional] The length of the minor axis of the ellipse. The default is 0.5. This is considered if the inputMode is 3 or 4.

sides
[int , optional] The number of sides of the ellipse. The default is 32.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the ellipse. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the ellipse. The default is 360.

close
[bool , optional] If set to True, arcs will be closed by connecting the last vertex to the first vertex. Otherwise, they will be left open.

direction
[list , optional] The vector representing the up direction of the ellipse. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the ellipse. This can be "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created ellipse

`static EllipseAll`(*origin=None*, *inputMode: int = 1*, *width: float = 2.0*, *length: float = 1.0*, *focalLength: float = 0.866025*, *eccentricity: float = 0.866025*, *majorAxisLength: float = 1.0*, *minorAxisLength: float = 0.5*, *sides: int = 32*, *fromAngle: float = 0.0*, *toAngle: float = 360.0*, *close: bool = True*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates an ellipse and returns all its geometry and parameters.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the ellipse. The default is None which results in the ellipse being placed at (0, 0, 0).

inputMode
[int , optional] The method by which the ellipse is defined. The default is 1. Based on the inputMode value, only the following inputs will be considered. The options are: 1. Width and Length (considered inputs: width, length) 2. Focal Length and Eccentricity (considered inputs: focalLength, eccentricity) 3. Focal Length and Minor Axis Length (considered inputs: focalLength, minorAxisLength) 4. Major Axis Length and Minor Axis Length (considered input: majorAxisLength, minorAxisLength)

width
[float , optional] The width of the ellipse. The default is 2.0. This is considered if the inputMode is 1.

length
[float , optional] The length of the ellipse. The default is 1.0. This is considered if the inputMode is 1.

focalLength
[float , optional] The focal length of the ellipse. The default is 0.866025. This is considered if the inputMode is 2 or 3.

eccentricity
[float , optional] The eccentricity of the ellipse. The default is 0.866025. This is considered if the inputMode is 2.

majorAxisLength
[float , optional] The length of the major axis of the ellipse. The default is 1.0. This is considered if the inputMode is 4.

minorAxisLength
[float , optional] The length of the minor axis of the ellipse. The default is 0.5. This is considered if the inputMode is 3 or 4.

sides
[int , optional] The number of sides of the ellipse. The default is 32.

fromAngle
[float , optional] The angle in degrees from which to start creating the arc of the ellipse. The default is 0.

toAngle
[float , optional] The angle in degrees at which to end creating the arc of the ellipse. The default is 360.

close
[bool , optional] If set to True, arcs will be closed by connecting the last vertex to the first vertex. Otherwise, they will be left open.

direction
[list , optional] The vector representing the up direction of the ellipse. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the ellipse. This can be "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

dictionary
A dictionary with the following keys and values: 1. "ellipse" : The ellipse (topologic_core.Wire) 2. "foci" : The two focal points (topologic_core.Cluster containing two vertices) 3. "a" : The major axis length 4. "b" : The minor axis length 5. "c" : The focal length 6. "e" : The eccentricity 7. "width" : The width 8. "length" : The length

`static EndVertex`(*wire*)

Returns the end vertex of the input wire. The wire must be manifold and open.

`static ExteriorAngles`(*wire*, *tolerance: float = 0.0001*, *mantissa: int = 6*) → list

Returns the exterior angles of the input wire in degrees. The wire must be planar, manifold, and closed.

Parameters

wire
[topologic_core.Wire] The input wire.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

mantissa
[int , optional] The length of the desired mantissa. The default is 6.

Returns

list
The list of exterior angles.

`static ExternalBoundary`(*wire*)

Returns the external boundary (cluster of vertices where degree == 1) of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

Returns

topologic_core.Cluster
The external boundary of the input wire. This is a cluster of vertices of degree == 1.

`static Fillet`(*wire*, *radius: float = 0*, *sides: int = 16*, *radiusKey: str = None*, *tolerance: float = 0.0001*, *silent: bool = False*)

Fillets (rounds) the interior and exterior corners of the input wire given the input radius. See https://en.wikipedia.org/wiki/Fillet_(mechanics)

Parameters

wire
[topologic_core.Wire] The input wire.

radius
[float] The desired radius of the fillet.

radiusKey
[str , optional] If specified, the dictionary of the vertices will be queried for this key to specify the desired fillet radius. The default is None.

sides
[int , optional] The number of sides (segments) of the fillet. The default is 16.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The filleted wire.

`static InteriorAngles`(*wire*, *tolerance: float = 0.0001*, *mantissa: int = 6*) → list

Returns the interior angles of the input wire in degrees. The wire must be planar, manifold, and closed. This code has been contributed by Yidan Xue.

Parameters

wire
[topologic_core.Wire] The input wire.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The list of interior angles.

`static Interpolate`(*wires: list*, *n: int = 5*, *outputType: str = 'default'*, *mapping: str = 'default'*, *tolerance: float = 0.0001*)

Creates *n* number of wires that interpolate between wireA and wireB.

Parameters

wireA
[topologic_core.Wire] The first input wire.

wireB
[topologic_core.Wire] The second input wire.

n
[int , optional] The number of intermediate wires to create. The default is 5.

outputType
[str , optional]

The desired type of output. The options are case insensitive. The default is "contour". The options are:

- "Default" or "Contours" (wires are not connected)
- "Raster or "Zigzag" or "Toolpath" (the wire ends are connected to create a continuous path)
- "Grid" (the wire ends are connected to create a grid).

mapping
[str , optional]

The desired type of mapping for wires with different number of vertices. It is case insensitive. The default is "default". The options are:

- "Default" or "Repeat" which repeats the last vertex of the wire with the least number of vertices
- "Nearest" which maps the vertices of one wire to the nearest vertex of the next wire creating a list of equal number of vertices.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Topology
The created interpolated wires as well as the input wires. The return type can be a topologic_core.Cluster or a topologic_core.Wire based on options.

`static Invert`(*wire*)

Creates a wire that is an inverse (mirror) of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

Returns

topologic_core.Wire
The inverted wire.

`static IsClosed`(*wire*) → bool

Returns True if the input wire is closed. Returns False otherwise.

Parameters

wire
[topologic_core.Wire] The input wire.

Returns

bool
True if the input wire is closed. False otherwise.

`static IsManifold`(*wire*, *silent: bool = False*) → bool

Returns True if the input wire is manifold. Returns False otherwise. A manifold wire is one where its vertices have a degree of 1 or 2.

Parameters

wire
[topologic_core.Wire] The input wire.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

bool
True if the input wire is manifold. False otherwise.

`static IsSimilar`(*wireA*, *wireB*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*) → bool

Returns True if the input wires are similar. Returns False otherwise. The wires must be closed.

Parameters

wireA
[topologic_core.Wire] The first input wire.

wireB
[topologic_core.Wire] The second input wire.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

bool
True if the two input wires are similar. False otherwise.

`static Length`(*wire*, *mantissa: int = 6*) → float

Returns the length of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

float
The length of the input wire. Test

`static Line`(*origin=None*, *length: float = 1*, *direction: list = [1, 0, 0]*, *sides: int = 2*, *placement: str = 'center'*)

Creates a straight line wire using the input parameters.

Parameters

origin
[topologic_core.Vertex , optional] The origin location of the box. The default is None which results in the edge being placed at (0, 0, 0).

length
[float , optional] The desired length of the edge. The default is 1.0.

direction
[list , optional] The desired direction (vector) of the edge. The default is [1, 0, 0] (along the X-axis).

sides
[int , optional] The desired number of sides/segments. The minimum number of sides is 2. The default is 2.

placement
[str , optional] The desired placement of the edge. The options are: 1. "center" which places the center of the edge at the origin. 2. "start" which places the start of the edge at the origin. 3. "end" which places the end of the edge at the origin. The default is "center".

Returns

topologic_core.Edge
The created edge

`static Miter`(*wire*, *offset: float = 0*, *offsetKey: str = None*, *tolerance: float = 0.0001*, *silent: bool = False*)

Fillets (rounds) the interior and exterior corners of the input wire given the input radius. See https://en.wikipedia.org/wiki/Fillet_(mechanics)

Parameters

wire
[topologic_core.Wire] The input wire.

offset
[float] The desired offset length of the miter along each edge.

offsetKey
[str , optional] If specified, the dictionary of the vertices will be queried for this key to specify the desired offset length. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The filleted wire.

`static Normal`(*wire*, *outputType='xyz'*, *mantissa=6*)

Returns the normal vector to the input wire. A normal vector of a wire is a vector perpendicular to it.

Parameters

wire
[topologic_core.Wire] The input wire.

outputType
[string , optional] The string defining the desired output. This can be any subset or permutation of "xyz". It is case insensitive. The default is "xyz".

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

list
The normal vector to the input face.

`static OrientEdges`(*wire*, *vertexA*, *transferDictionaries=False*, *tolerance=0.0001*)

Returns a correctly oriented head-to-tail version of the input wire. The input wire must be manifold.

Parameters

wire
[topologic_core.Wire] The input wire.

vertexA
[topologic_core.Vertex] The desired start vertex of the wire.

transferDictionaries
[bool , optional] If set to True, the dictionaries of the original wire are transfered to the new wire. Otherwise, they are not. The default is False.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The oriented wire.

`static Planarize`(*wire*, *origin=None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns a planarized version of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

tolerance
[float, optional] The desired tolerance. The default is 0.0001.

origin
[topologic_core.Vertex , optional] The desired origin of the plane unto which the planar wire will be projected. If set to None, the centroid of the input wire will be chosen. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

Returns

topologic_core.Wire
The planarized wire.

`static Project`(*wire*, *face*, *direction: list = None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Creates a projection of the input wire unto the input face.

Parameters

wire
[topologic_core.Wire] The input wire.

face
[topologic_core.Face] The face unto which to project the input wire.

direction
[list, optional] The vector representing the direction of the projection. If None, the reverse vector of the receiving face normal will be used. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The projected wire.

`static Rectangle`(*origin=None*, *width: float = 1.0*, *length: float = 1.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Creates a rectangle.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the rectangle. The default is None which results in the rectangle being placed at (0, 0, 0).

width
[float , optional] The width of the rectangle. The default is 1.0.

length
[float , optional] The length of the rectangle. The default is 1.0.

direction
[list , optional] The vector representing the up direction of the rectangle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the rectangle. This can be “center”, “lowerleft”, “upperleft”, “lowerright”, “upperright”. It is case insensitive. The default is “center”.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created rectangle.

`static RemoveCollinearEdges`(*wire*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*, *silent: bool = False*)

Removes any collinear edges in the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The created wire without any collinear edges.

`static Reverse`(*wire*, *transferDictionaries=False*, *tolerance: float = 0.0001*)

Creates a wire that has the reverse direction of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

transferDictionaries
[bool , optional] If set to True the dictionaries of the input wire are transferred to the new wire. Othwerwise, they are not. The default is False.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The reversed wire.

`static Roof`(*face*, *angle: float = 45*, *boundary: bool = True*, *tolerance: float = 0.001*)

Creates a hipped roof through a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com> This algorithm depends on the polyskel code which is included in the library. Polyskel code is found at: https://github.com/Botffy/polyskel

Parameters

face
[topologic_core.Face] The input face.

angle
[float , optioal] The desired angle in degrees of the roof. The default is 45.

boundary
[bool , optional] If set to True the original boundary is returned as part of the roof. Otherwise it is not. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.001. (This is set to a larger number as it was found to work better)

Returns

topologic_core.Wire
The created roof. This method returns the roof as a set of edges. No faces are created.

`static Simplify`(*wire*, *method='douglas-peucker'*, *tolerance=0.0001*, *silent=False*)

Simplifies the input wire edges based on the selected algorithm: Douglas-Peucker or Visvalingam–Whyatt.

Parameters

wire
[topologic_core.Wire] The input wire.

method
[str, optional] The simplification method to use: 'douglas-peucker' or 'visvalingam-whyatt' or 'reumann-witkam'. The default is 'douglas-peucker'.

tolerance
[float , optional] The desired tolerance. If using the douglas-peucker method, edge lengths shorter than this amount will be removed. If using the visvalingam-whyatt method, triangulare areas less than is amount will be removed. If using the Reumann-Witkam method, the tolerance specifies the maximum perpendicular distance allowed between any point and the current line segment; points falling within this distance are discarded. The default is 0.0001.

silent
[bool , optional] If set to True, no error and warning messages are printed. Otherwise, they are. The default is False.

Returns

topologic_core.Wire
The simplified wire.

`static Skeleton`(*face*, *boundary: bool = True*, *tolerance: float = 0.001*)

Creates a straight skeleton. This method is contributed by xipeng gao <gaoxipeng1998@gmail.com> This algorithm depends on the polyskel code which is included in the library. Polyskel code is found at: https://github.com/Botffy/polyskel

Parameters

face
[topologic_core.Face] The input face.

boundary
[bool , optional] If set to True the original boundary is returned as part of the roof. Otherwise it is not. The default is True.

tolerance
[float , optional] The desired tolerance. The default is 0.001. (This is set to a larger number as it was found to work better)

Returns

topologic_core.Wire
The created straight skeleton.

`static Spiral`(*origin=None*, *radiusA: float = 0.05*, *radiusB: float = 0.5*, *height: float = 1*, *turns: int = 10*, *sides: int = 36*, *clockwise: bool = False*, *reverse: bool = False*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a spiral.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the spiral. The default is None which results in the spiral being placed at (0, 0, 0).

radiusA
[float , optional] The initial radius of the spiral. The default is 0.05.

radiusB
[float , optional] The final radius of the spiral. The default is 0.5.

height
[float , optional] The height of the spiral. The default is 1.

turns
[int , optional] The number of turns of the spiral. The default is 10.

sides
[int , optional] The number of sides of one full turn in the spiral. The default is 36.

clockwise
[bool , optional] If set to True, the spiral will be oriented in a clockwise fashion. Otherwise, it will be oriented in an anti-clockwise fashion. The default is False.

reverse
[bool , optional] If set to True, the spiral will increase in height from the center to the circumference. Otherwise, it will increase in height from the conference to the center. The default is False.

direction
[list , optional] The vector representing the up direction of the spiral. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the spiral. This can be “center”, “lowerleft”, “upperleft”, “lowerright”, “upperright”. It is case insensitive. The default is “center”.

Returns

topologic_core.Wire
The created spiral.

`static Split`(*wire*) → list

Splits the input wire into segments at its intersections (i.e. at any vertex where more than two edges meet).

Parameters

wire
[topologic_core.Wire] The input wire.

Returns

list
The list of split wire segments.

static Square(*origin=None*, *size: float = 1.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a square.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the square. The default is None which results in the square being placed at (0, 0, 0).

size
[float , optional] The size of the square. The default is 1.0.

direction
[list , optional] The vector representing the up direction of the square. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the square. This can be "center", "lowerleft", "upperleft", "lowerright", "upperright". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created square.

static Squircle(*origin=None*, *radius: float = 0.5*, *sides: int = 121*, *a: float = 2.0*, *b: float = 2.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *angTolerance: float = 0.1*, *tolerance: float = 0.0001*)

Creates a Squircle which is a hybrid between a circle and a square. See https://en.wikipedia.org/wiki/Squircle

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the squircle. The default is None which results in the squircle being placed at (0, 0, 0).

radius
[float , optional] The desired radius of the squircle. The default is 0.5.

sides
[int , optional] The desired number of sides of the squircle. The default is 121.

a
[float , optional] The "a" factor affects the x position of the points to interpolate between a circle and a square. A value of 1 will create a circle. Higher values will create a more square-like shape. The default is 2.0.

b
[float , optional] The "b" factor affects the y position of the points to interpolate between a circle and a square. A value of 1 will create a circle. Higher values will create a more square-like shape. The default is 2.0.

direction
[list , optional] The vector representing the up direction of the circle. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the circle. This can be "center", "lowerleft", "upperleft", "lowerright", or "upperright". It is case insensitive. The default is "center".

angTolerance
[float , optional] The desired angular tolerance. The default is 0.1.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created squircle.

`static Star`(*origin=None*, *radiusA: float = 0.5*, *radiusB: float = 0.2*, *rays: int = 8*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a star.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the star. The default is None which results in the star being placed at (0, 0, 0).

radiusA
[float , optional] The outer radius of the star. The default is 1.0.

radiusB
[float , optional] The outer radius of the star. The default is 0.4.

rays
[int , optional] The number of star rays. The default is 8.

direction
[list , optional] The vector representing the up direction of the star. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the star. This can be "center", "lowerleft", "upperleft", "lowerright", or "upperright". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created star.

`static StartEndVertices`(*wire*) → list

Returns the start and end vertices of the input wire. The wire must be manifold and open.

`static StartVertex`(*wire*)

Returns the start vertex of the input wire. The wire must be manifold and open.

static Trapezoid(*origin=None*, *widthA: float = 1.0*, *widthB: float = 0.75*, *offsetA: float = 0.0*, *offsetB: float = 0.0*, *length: float = 1.0*, *direction: list = [0, 0, 1]*, *placement: str = 'center'*, *tolerance: float = 0.0001*)

Creates a trapezoid.

Parameters

origin
[topologic_core.Vertex , optional] The location of the origin of the trapezoid. The default is None which results in the trapezoid being placed at (0, 0, 0).

widthA
[float , optional] The width of the bottom edge of the trapezoid. The default is 1.0.

widthB
[float , optional] The width of the top edge of the trapezoid. The default is 0.75.

offsetA
[float , optional] The offset of the bottom edge of the trapezoid. The default is 0.0.

offsetB
[float , optional] The offset of the top edge of the trapezoid. The default is 0.0.

length
[float , optional] The length of the trapezoid. The default is 1.0.

direction
[list , optional] The vector representing the up direction of the trapezoid. The default is [0, 0, 1].

placement
[str , optional] The description of the placement of the origin of the trapezoid. This can be "center", or "lowerleft". It is case insensitive. The default is "center".

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Wire
The created trapezoid.

static VertexByDistance(*wire*, *distance: float = 0.0*, *origin=None*, *tolerance=0.0001*)

Creates a vertex along the input wire offset by the input distance from the input origin.

Parameters

wire
[topologic_core.Wire] The input wire.

distance
[float , optional] The offset distance. The default is 0.

origin
[topologic_core.Vertex , optional] The origin of the offset distance. If set to None, the origin will be set to the start vertex of the input edge. The default is None.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

topologic_core.Vertex
The created vertex.

`static VertexByParameter`(*wire*, *u: float = 0*)

Creates a vertex along the input wire offset by the input *u* parameter. The wire must be manifold.

Parameters

wire
[topologic_core.Wire] The input wire.

u
[float , optional] The *u* parameter along the input topologic Wire. A parameter of 0 returns the start vertex. A parameter of 1 returns the end vertex. The default is 0.

Returns

topologic_core.Vertex
The vertex at the input u parameter

`static VertexDistance`(*wire*, *vertex*, *origin=None*, *mantissa: int = 6*, *tolerance: float = 0.0001*)

Returns the distance, computed along the input wire of the input vertex from the input origin vertex.

Parameters

wire
[topologic_core.Wire] The input wire.

vertex
[topologic_core.Vertex] The input vertex

origin
[topologic_core.Vertex , optional] The origin of the offset distance. If set to None, the origin will be set to the start vertex of the input wire. The default is None.

mantissa
[int , optional] The desired length of the mantissa. The default is 6.

tolerance
[float , optional] The desired tolerance. The default is 0.0001.

Returns

float
The distance of the input vertex from the input origin along the input wire.

`static Vertices`(*wire*) → list

Returns the list of vertices of the input wire.

Parameters

wire
[topologic_core.Wire] The input wire.

Returns

list
The list of vertices.

topologicpy.version module

Module contents

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z