Python API Concise Reference Guide

Version 2022.06-SP2, December 2022

SYNOPSYS®

Copyright and Proprietary Information Notice

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

www.synopsys.com

Feedback

Contents

1

Introduction

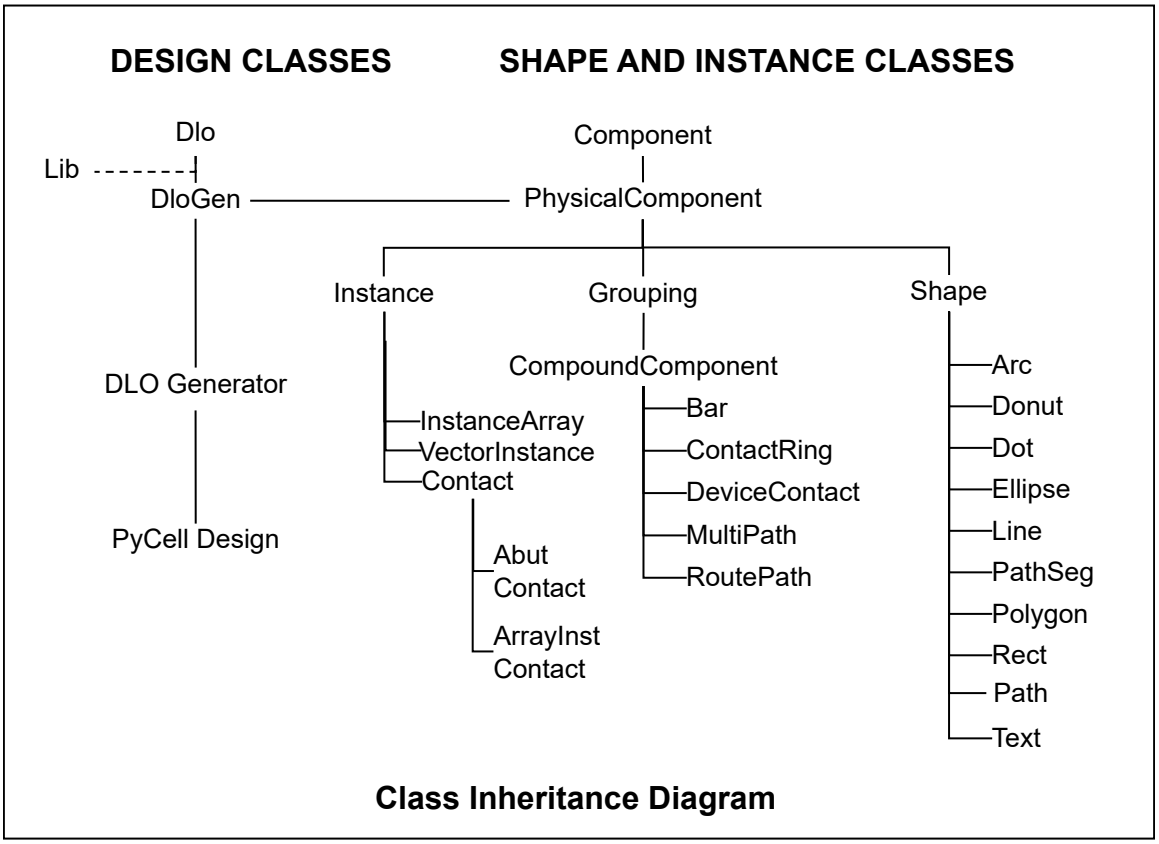
This guide provides a concise description of all of the Python API commands. While this guide is designed to be a handy quick reference to the full Python API, it is not meant to be a replacement for the much more in-depth reference material provided in the complete Python API documentation. Instead, this guide is meant to serve as a quick refresher, so that basic information about a certain class or method can be quickly located. This guide should be used in conjunction with the complete Python API Reference manual document.

How to Use this Reference Guide

This Reference Guide contains a section for each of the classes defined for the Python API. A brief description of the overall functionality of the class is given, along with very brief descriptions of each method defined for that class. These methods are listed in alphabetical order, except for any creation methods for the class, which are always listed at the beginning. Each class method description includes the listing of the parameter keywords and expected parameter types, along with a short description of the functionality of that method. The return value for each class method is also described. In addition, any attributes or properties for the class are briefly described.

Feedback

Class Organization



Class Inheritance Diagram

Technology Classes

- Layer
- LayerMaterial
- ShapeFilter, Grid
- PhysicalRule, RuleProperty
- Tech

Connectivity Classes

- SignalType, TermType, Net
- BusNet, BundleNet, Pin, Term
- BusTerm, BundleTerm, InstPin

Feedback

- InstTerm, ShapeRef, Topology
- RouteTarget

Basic Geometric Classes

- Point
- Range, Segment
- Box
- Direction
- Location
- Orientation, Transform
- PathStyle, GapStyle

Utility Classes

- ParamArray, ParamSpecArray
- PropSet, PointList
- Log, Unique, NameMapper
- Numeric, cFloat, AttrDict,
- OrderedDict, ParamDictSpec, CDF, DPL,
- Grid, SnapType, Fill, CrossOver, Marker
- DrcSummary

2

Class Methods

Design Classes

The design classes include the Dlo, DloGen, and Lib classes.

Dlo

Base class for DloGen and all design related classes. This is an abstract base class, which is only used for class derivation, and cannot be directly created. DLO is an acronym for “Dynamic Layout Object”.

exists(string *dloName* **)**

returns True if the *dloName* Dlo design object exists, and False otherwise

findComp(string *name*=“”)

returns physical component having this *name* in current design

findCompRef(string *name*)

returns the PhysicalCompRef physical component reference object in the current design having this *name* in the current design

findPin(string *name*=“”)

returns the Pin object within Dlo which has the same name as the *name* parameter

findTerm(string *name*=“”)

returns the Term object within Dlo which has the same name as the *name* parameter

getBBox(ShapeFilter *filter*=ShapeFilter())

returns a bounding box for all physical components, using any specified layers

getCellType()

returns the cell type for this Dlo design object

Feedback

getComps()

returns a uniform list of all components and groupings in this DloGen object

getLeafComps()

returns a uniform list of all leaf-level components which are contained in this Dlo object

getLib()

returns the open library associated with this Dlo

getLpps()

returns a uniform list of layers for this Dlo

getName()

returns string containing the library name, cell name and view name for this Dlo

getParams(ParamArray *params*=None, bool *all*=True)

returns the ParamArray object which provides the explicit parameters and values which were used when this Dlo object was created

getPins()

returns a uniform list of all of the Pin objects which are contained in this Dlo design object

getProps()

returns PropSet Python dictionary-like object containing properties for this Dlo

getTech()

returns current Tech technology object associated with this Dlo design object

getTermOrder()

returns ordered terminal name list, which defines terminal connectivity by position

getTerms()

returns a uniform list of all of the Term objects which are contained in this Dlo design object

getViewType()

returns the view type associated with this Dlo design object

isSubMaster()

returns True if this Dlo design is a SubMaster design, and False otherwise

Feedback

isSuperMaster()

returns True if this Dlo design is a SuperMaster design, and False otherwise

save()

saves this Dlo design using the library name, cell name and view name used when this Dlo object was created

saveAs(string *libName*, string *cellName*, string *viewName* = “”)

saves this Dlo design using the passed library name, cell name and view name

setCustomLocation(Location *loc*, Point *point*)

sets a global custom location *loc* with a position specified by the *point* parameter. This location is over the cell object.

setTermOrder()

ordered list of terminal names defines terminal connectivity by position

Attributes:

- **props** – returns any property-value pairs defined for this Dlo design object
- **tech –** returns read-only Tech object associated with this Dlo design object

---

DloGen

Base class for all DLO design generators. This is an abstract base class, only used for class derivation, and cannot be directly created. The PyCell™ author must provide the virtual methods which actually create design layout for the parameterized cell.

addPin(string *pinName*, string *termName*, Box *box*, Layer *layer*)

addPin(string *pinName*, string *termName*, Box *box*, LayerList *layers*)

adds pin having specified name for the specified terminal to design represented by this DloGen object; creates geometries for pin on specified layers.

addTerm(string *name*, TermType *termType*=TermType.INPUT_OUTPUT)

adds a terminal with the specified name to this DloGen design object

currentDloGen()

returns current DloGen design object; used within non-DloGen derived classes

dbu2uu(int *value*)

returns user unit value corresponding to specified database unit value

Feedback

dbu2uuArea(int *value*)

returns user unit value corresponding to specified square database unit value

fgAnd(self, ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None)

performs a logical AND operation for lists of physical components *comps1* and *comps*2, by selecting those polygon areas which are in both physical components.

fgDeriveLayer(self, ulist[PhysicalComponent] *comps*, string *derivationScript*, Layer *layer*, ShapeFilter *filter*=ShapeFilter(), float *grid*=None)

runs the internal geometry engine to generate derived layer shapes according to the *derivationScript* parameter which contains a sequence of DRC commands to be executed, while the *layer* parameter specifies the layer on which to create the resulting shapes.

fgNot(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None)

performs a logical NOT operation for lists of physical components *comps1* and *comps*2, by selecting those polygon areas contained in the *comps1* physical component which are not contained in the *comps2* physical component.

fgOr(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None)

performs a logical OR operation for lists of physical components *comps1* and *comps*2, by selecting those polygon areas which are in either list of physical components.

fgSize(ulist[PhysicalComponent] *comps*, ShapeFilter *filter*, Coord *sizeValue*, Layer *resultLayer,* float *grid*=None)

expands or shrinks all polygon areas contained in this list of physical components, according to the *sizeValue* parameter value.

fgXor(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None)

performs a logical eXclusive-OR operation for lists of physical components *comps1* and *comps*2, by selecting those polygon areas which are in either list of physical components, but not in both lists of physical components.

Feedback

getComp(int *index*)

returns component in this DloGen object at the specified *index* value

getInsts(Box *box*=None, startLevel=0, stopLevel=0)

returns a list of all Instance / InstanceArray / VectorInstance and/or InstanceRef / InstanceArrayRef / VectorInstanceRef objects within this DloGen design object which are on the specified layer and overlap the specified box

getNets()

returns uniform list of all nets in this DloGen object

getShapes(Layer *layer*=None, Box *box*=None, startLevel=0, stopLevel=0)

returns a list of all Shape and/or ShapeRef objects within this DloGen design object which are on the specified layer and overlap the specified box

makeCompName(string *prefix*)

generates unique name for physical component within this DloGen design object

makeGrouping(string *name*='')

creates Grouping object containing all physical components in this DloGen object

makeNetName(string *prefix*)

generates unique name for net within this DloGen design object

makePinName(string *prefix*)

generates unique name for pin within this DloGen design object

makePinNameSpecNum(*string prefix, unsigned int num*)

generates unique name with user specified pin number for pin within this DloGen design object.

makeTermName(string *prefix*)

generates unique name for terminal within this DloGen design object

mirrorX(Coord *yCoord*=0)

mirrors all physical components about the X coordinate axis

mirrorY(Coord *xCoord*=0)

mirrors all physical components about the Y coordinate axis

moveBy(Coord *dx*, Coord *dy*)

moves all physical components *dx* units in x-direction and *dy* units in y-direction

Feedback

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter())

moves all physical components, so that specified *destination* point becomes handle point location for the bounding box for this physical component

moveTowards(Direction *dir*, Coord *distance*)

moves all physical components by specified distance in the specified direction

rotate90(Point *origin*=None)

rotates all physical components 90 degrees, in counter-clockwise direction

rotate180(Point *origin*=None)

rotates all physical components 180 degrees, in counter-clockwise direction

rotate270(Point *origin*=None)

rotates all physical components 270 degrees, in counter-clockwise direction

setOrigin(Point *origin*)

moves all physical components so that specified point becomes origin for design

transform(Transform *trans*)

applies the transform *trans* passed as a parameter to all physical components

uu2dbu(double *value*)

returns database unit value corresponding to specified user unit value

uu2dbuArea(double *value*)

returns database unit value corresponding to specified square user unit value

withNewDlo(object *callback*, string *lname*, string *cname*, string *vname*=None)

creates DLO for regression testing, using specified library, cell and view names

Attributes:

- **AttrType** – display attribute type for this DloGen design object

PyCell author provided methods:

The parameterized cell author must provide methods which override these virtual methods defined for the DloGen base class

defineParamSpecs(ParamSpecArray *specs*)

defines parameters, default values and constraints for this parameterized cell (must be defined using “@classmethod” or “@staticmethod” Python syntax)

Feedback

genLayout()

generates actual physical layout for this parameterized cell

genTopology()

generates topology for the devices in this parameterized cell

setupParams(ParamArray *params*)

extracts the values for parameters entered by user of this parameterized cell

sizeDevices()

sizes the devices, before generating layout, for this parameterized cell

---

VersionedDloGen

This class is used to provide different versions of the Python source code for the same PyCell. The PyCell developer uses this class to create PyCells which use different versions of the source code to generate different versions of the layout for the PyCell. The PyCell user can then select a particular version of the Python source code by means of a special code versioning parameter for the PyCell.

Note that there are no methods for this class It is sufficient to derive the PyCell class from this class and use the built-in naming convention for versioned class names.

Lib

Class used to represent the OpenAccess library which is used to read and write PyCell design objects.

attachTechLib(string *techLibName*)

attaches Santana technology library to this Lib object

close(string *name*, string *path*)

closes library and destroys associated Lib object

create(string *name*, string *path*)

create(string *name*, string *path*)

creates library using specified name and path; raises exception if cannot be created

definePcell(object *cls*, string *cellName*, string *ViewName*='layout', string *viewType*=None, string *cellType* = None)

creates a parameterized cell in this library, using specified DloGen class

Feedback

getName()

returns the name of this library

getPath()

returns the file path for this library

getProps(*cellName*=None, *viewName*=None, *readonly*=True)

returns PropSet Python dictionary-like object containing properties for this library; if *cellName* and/or *viewName* is set, then cell or cell view properties are returned.

getTech(*cnTechVersion*=None)

returns Santana Tech object associated with this library

getTechFilePath()

returns file path for the current Santana technology file attached to this library

getTechLibName()

returns name of current Santana technology library attached to this library

installTechFile(string *techFilePath*, bool *withCoreDlos*=True, bool *forceBinary*=False, bool *createTechDB*=True, string *displayFilePath*=None)

copies technology information from Santana technology file into this library

loadPcells(string *cpkgID*)

loads parameterized cells stored in package into this library

open(string *name*, string *path*=”)

opens library using specified name and path; raises exception if cannot be opened.

updatePcell(string *cellName*, string *viewName* = 'layout')

updates existing parameterized cell in this library, using current bindings

Attributes:

- **props** – returns any property-value pairs defined for this Lib object
- **tech** – returns current technology object for this library

Feedback

Shape and Instance Classes

The shape and instance classes represent all of the shapes and components which can be manufactured in a chip layout. In addition to the basic Shape class objects consisting of a single component drawn on a single layer, the Instance and Grouping classes provide multiple component layout objects which are drawn on multiple layers. These classes include higher-level objects such as contacts, bars, multi-paths, routes and contact rings.

PhysicalComponent

The base class from which all physical components and shapes are derived. This is an abstract base class, which is only used for class derivation, and cannot be directly created.

abut(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*, bool *align*=True, ShapeFilter *refFilter*=None)

moves physical component in the specified direction, until its bounding box just touches the bounding box for the specified *refComp* physical component

alignEdge(Direction *dir*, PhysicalComponent *refComp*, Direction *refDir*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Coord *offset*=None)

aligns edges of bounding boxes of these two physical components, using the specified directions to perform this one-dimensional alignment operation

alignEdgeToPoint(Direction *dir*, Point *point*, ShapeFilter *filter*=ShapeFilter())

aligns edge of bounding box of physical component to specified *point*

alignLocation(Location *loc*, PhysicalComponent *refComp*, Location *refLoc*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Point *offset*=None)

aligns locations of bounding boxes of these two physical components, using the specified locations to perform this two-dimensional alignment operation

alignLocationToPoint(Location *loc*, Point *point*, ShapeFilter *filter*=ShapeFilter())

aligns location of bounding box of physical component to specified *point*

clone()

virtual method to be overridden in derived class, to return cloned copy

destroy()

destroys this physical component, as well as underlying OpenAccess object

find(string *name*=”)

returns physical component having specified name in current design

fgAbut(Direction *dir*, PhysicalComponent *refComp*, LayerList *abutLayers*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, bool *align*=True, ShapeFilter *refFilter*=None, dict *options*=None, float *grid*=None)

combines functionality of **fgPlace**() and **fgFill**() methods; this component is placed next to *refComp* component, abutting them on the *abutLayers* list of layers. New shapes will be created to fill any gaps between components on these layers.

fgAddEnclosingPolygon(Layer *layer*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, float *grid*=None)

creates enclosing polygon on the specified layer, by using minimum enclosure technology design rules. Note that non-rectangular polygons can be generated.

fgAnd(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilter*=None, float *grid*=None)

performs logical and operation for polygon shapes contained in each physical component, on specified layers; shapes are generated on the *resultLayer*.

fgDeriveLayer(string *derivationScript*, Layer *layer*, ShapeFilter *filter*=ShapeFilter(), float *grid*=None)

runs DRC program to check DRC correctness for this physical component, using design rules in technology file attached to current design.

fgEnclose(LayerList *layers*, ShapeFilter *filter*=ShapeFilter(), dict *options*=None, float *grid*=None)

creates rectangles that enclose the physical component on every layer in the given layer list. All minimum enclose design rules are considered to produce the DRC correct set of enclosing rectangles.

fgExtend(PhysicalComponent *comp*, ShapeFilter *filter*=ShapeFilter(), dict *options*=None, float *grid*=None)

for a single Rect object or a grouping of Rect objects, extends the rectangles in order to satisfy all minimum extension design rules between the rectangles and other shapes overlapping them.

fgFill(PhysicalComponent *comp*, Layer *layer*, Direction *dir*, Coord *divider,* float *grid*=None)

performs DRC-correct fill operation between two components, in given direction; *divider* specifies dividing line between two components for this fill operation.

fgMerge(Layer *resultLayer*=None, ShapeFilter *filter*=None, float *grid*=None)

merges all polygon shapes contained on single layer for the physical component; shapes are generated on the *resultLayer*.

Feedback

fgMinSpacing(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, *align*=True, ShapeFilter *refFilter*=None, dict *options*=None)

returns minimum “DRC clean” distance between physical components

fgNot(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilter*=None, float *grid*=None)

performs logical not operation for polygon shapes contained in each physical component, on specified layers; shapes are generated on the *resultLayer*.

fgOr(PhysicalComponent *comp*, Layer *resultLayer*,

ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilter*=None, float *grid*=None) performs logical or operation for polygon shapes contained in each physical component, on specified layers; shapes are generated on the *resultLayer*.

fgPlace(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, bool *align*=True, ShapeFilter *refFilter*=None, dict *options*=None, float *grid*=None)

performs “smart placement” of physical component, by moving it in the specified direction, such that it has minimal distance spacing from the *refComp* physical component. This spacing is determined from current set of technology design rules.

fgSize(ShapeFilter *filter*, Coord *sizeValue*, Layer *resultLayer,* float *grid*=None)

expands or shrinks polygon shapes contained in *comp* physical component, based on value of *sizeValue* parameter; shapes are generated on the *resultLayer*.

fgXor(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilte*r=None, float *grid*=None)

performs logical exclusive-or operation for polygon shapes contained in each physical component, on specified layers; shapes are generated on the *resultLayer*.

getBBox(ShapeFilter filter=ShapeFilter())

returns bounding box for this physical component, using any specified layers

getCompOwner()

returns owner of this physical component; by default, this is current DloGen object.

getDlo()

returns Dlo or DloGen design object which contains this physical component

Feedback

getLocationPoint(Location *loc*, Point *default*=None)

returns a point corresponding to the *loc* location parameter which can be one of nine standard (pre-defined) locations, or a custom location previously created by calling setCustomLocation().

getName()

returns name for this physical component

getProps()

returns PropSet Python dictionary-like object containing properties for this object

getSpacing(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter* = None)

uses bounding boxes to calculate current distance between this physical component and *refComp* physical component in the specified *dir* direction.

keepRelativePosition(IPhysicalComponent *comp*, bool *keep* = True)

keeps the alignment of physical components throughout PyCell evaluation.

makeArray(Coord *dX*, Coord *dY*, unsigned *numRows*, unsigned *numCols*, *baseName*='', *name*='', bool *rowMajor*=True)

creates array of components, as *numRows* rows by *numCols* columns; *dX* and *dY* specify spacing, while *baseName* is used to name components

mirrorX(Coord *yCoord*=0)

mirrors physical component about the X coordinate axis

mirrorY(Coord *xCoord*=0)

mirrors physical component about the Y coordinate axis

moveBy(Coord *dx*, Coord *dy*)

moves physical component *dx* units in x-direction and *dy* units in y-direction

moveTo(Coord *x*, Coord *y*, Location *loc*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter())

moves physical component, so that specified point coordinates become the *loc* point location for the bounding box of this physical component

moveTowards(Direction *dir*, Coord *d*)

moves physical component by specified distance in the specified direction

overlaps(Box *box*)

returns True if bounding box for this physical component overlaps specified box

Feedback

place(Direction *dir*, PhysicalComponent *refComp*, Coord *distance*, ShapeFilter *filter*=ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None)

performs explicit placement of physical component, by moving it in the specified direction, until it is the specified distance from the *refComp* physical component.

rotate90(Point *origin*=None)

rotates physical component 90 degrees in counter-clockwise direction

rotate180(Point *origin*=None)

rotates physical component 180 degrees in counter-clockwise direction

rotate270(Point *origin*=None)

rotates physical component 270 degrees in counter-clockwise direction

setCustomLocation(Location *loc*, Point *point*)

associates *loc* custom location for this physical component with a position specified by *point* parameter.

setName(string *name*)

sets name for this physical component

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

snaps to manufacturing grid points on the layers specified by the *filter* parameter; ensures that components are aligned to manufacturing grid points.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

snaps X coordinate of lower-left vertex point of bounding box for this component

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

snaps Y coordinate of lower-left vertex point of bounding box for this component

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter())

snaps lower-left vertex point of bounding box for component in the *dir* direction

transform(Transform *trans*)

applies the transform *trans* passed as a parameter to this physical component

Attribute:

- **props** – returns any property-value pairs defined for this PhysicalComponent

Feedback

Instance

The base class for all instances of DLO design objects.

Instance(string *dloName*, *params*=None, NodeSpec *nodeSpec*=None, string *name*='', Transform *trans*=None, bool *checkParams*=False)

creates instance of DLO master, using specified parameters and instance name

clone()

returns a cloned copy of this Instance object

find(string *name*='')

returns Instance object having specified name in current design

findInstPin(string *name*='')

returns instance pin having specified name for this Instance object

findInstTerm(string *name*='')

returns instance terminal having specified name for this Instance object

flatten(bool *promoteNames*=False, bool *promotePins*=False, bool *promoteBlockages*=False)

flattens this Instance to primitive shapes, and returns them as a Grouping

getBBox(ShapeFilter *filter*=ShapeFilter())

returns the bounding box for this Instance, using layers specified in *filter* parameter

getBit(unsigned int *index*)

for a single-bit instance, always returns this Instance object

getDefaultParams(ParamArray *params*=None)

returns array of default parameters used when this Instance object was created

getDloName()

returns master design name, as string "<libName>/<cellName>/<viewName>"

getInstPins()

returns list of all instance pins for this instance

getInstTerms()

returns list of all instance terminals for this instance

getMaster(bool *buildPyObj*=False)

returns the master design object used to create this Instance object

getName()

returns the name of this Instance object

getNumBits()

returns the number of bits of this Instance.

getOrientation()

returns Orientation object for this instance

getOrigin()

returns the point which is the origin location for this Instance object

getCompRefs()

returns uniform list of all physical component reference ojects inside this Instance

getParams(ParamArray *params*=None, bool *all*=True)

returns array of explicit parameters used when this Instance object was created

getParamSpecs()

returns parameter specifications used when this Instance object was created

getTransform()

returns Transform object for this Instance object

setConnectivity(self, NodeSpec *nodeSpec* bool strict=True)

Defines connectivity by specifying connections between terminals and nets. The *nodeSpec* parameter can be None (for no connectivity), Python dictionary associating nets with instance terminals, list of nets, or a single net. When the optional parameter strict is set to False, NodeSpec does not have to specify nets for all terms.

setMaster(string *libName*="", string *viewName*="", string *cellName*="")

sets the master design for this Instance object

setName(string *name*)

sets the name for this Instance object

setOrientation(Orientation *orient*)

sets Orientation object for this instance

Feedback

setOrigin(Point *point*)

sets specified point to be origin location for this Instance object

setParams(ParamArray *params*, bool *checkParams*=False)

set the parameter values for this Instance object

setTransform(Transform *trans*)

sets Transform object for this Instance object

Attribute:

- **name** – returns name for this Instance object (settable attribute)
- **AttrType** – display attribute type for this Instance object

---

InstanceArray

The InstanceArray class is derived from the base Instance class, and used to represent an array of instances of a single DLO master design. This class is typically used for memory efficiency purposes. For example, it can be used to represent an array of individual memory cells in a physical layout design.

InstanceArray(string *dloName*, Coord *dX*, Coord *dY*, unsigned int *numRows*, unsigned int *numCols*, ParamArray *params*=None, NodeSpec *nodeSpec*=None, string *name*=”, Transform *trans*=None, bool *checkParams*=False)

creates an array of instances of this DLO master, using the specified parameters, number of rows and columns, and the dX and dY array spacing between instances

clone()

returns a cloned copy of this InstanceArray object

find(string *name*=”)

returns InstanceArray object having specified name in current design

flatten()

flattens this InstanceArray to primitive shapes, and returns them as a Grouping

getDX()

returns offset spacing between columns in the X direction

getDY()

returns offset spacing between rows in the Y direction

Feedback

getMemberBBox(unsigned int *row*, unsigned int *col*, ShapeFilter *filter* = ShapeFilter())

returns the bounding box for instance at specified row and column in array; if ShapeFilter is used, only those layers are used in bounding box calculations

getMemberRefs()

returns list of members of this InstanceArray being used as reference objects

getMemberTransform(unsigned int *row*, unsigned int *col*)

returns transform for instance at specified row and column in array

getNumCols()

returns number of columns for this instance array

getNumRows()

returns number of rows for this instance array

setDX(Coord *dX*)

sets offset spacing between columns in the X direction

setDY(Coord *dY*)

sets offset spacing between rows in the Y direction

setName(string *name*)

sets the name for this instance array

setNumCols()

sets number of columns for this instance array

setNumRows()

sets number of rows for this instance array

Attribute:

- **name** – returns name for this Instance object (settable attribute)

VectorInstance

A vector instance represents several copies of the instance master with a range of index numbers to differentiate them. The VectorInstance class is derived from the Instance class. Vector instances have vector names, for example, I[1:3].

VectorInstance(string *dloName*, unsigned int *startIndex*, unsigned int *stopIndex*, ParamArray *params* = None, NodeSpec *nodeSpec* = None, string *baseName* = “”, Transform *trans* = None, bool *checkParams* = False)

creates a vector instance object using the the parameters:

clone(NameMapper *nameMap* = NameMapper(), NameMapper *netMap* = NameMapper())

returns a cloned copy of this vector instance object.

find(string *name* = “”)

returns the vector instance object having this name in the current DloGen design.

flatten(bool *promoteNames* = False, bool *promotePins* = False)

flattens this vector instance into the lowest-level primitive layout shapes which are contained in this vector instance.

getBaseName()

returns the base part of the name of this vector instance (without bit indexing).

getBit(unsigned int *index*)

returns a vector instance bit by its index.

getCompRefs()

returns a uniform list of all of the PhysicalCompRef objects which are references to physical components contained inside this vector instance.

getNumBits()

returns the number of bits of this vector instance.

getStartIndex()

returns the starting index of this vector instance bits.

getStopIndex()

returns the stopping index of this vector instance bits.

setBaseName(string *baseName*)

sets the base part of this vector instance name.

setIndexRange(unsigned int *startIndex*, unsigned int *stopIndex*)

sets the starting and stopping indexes of this vector instance bits.

Feedback

Contact

The Contact class provides a connection between specified interconnect layers. The Contact DLO is directly derived from the Instance class. Any such Contact objects will be constructed to meet DRC and electrical design rules. Note that the Contact object will be constructed using one or more cuts, which are used to improve manufacturing yield; the minimum number of cuts can be set through the “setMinCuts()” class method.

Contact(Layer *layer1*, Layer *layer2*, NodeSpec *nodeSpec*=None, Direction *routeDir1*=None, Direction *routeDir2*=None, Point *point1*=INVALID, Point *point2*=INVALID, Direction *anchor*=CENTER, LayerList *addLayers*=None, string *name*=”)

creates a Contact object which connects the two specified layers. The specified points are used to define the reference box for the Contact object, while *addLayers* is used to specify any additional layers to be used to construct the Contact.

clone()

returns a cloned copy of this Contact object

getLayer1()

returns the first connection layer for this contact

getLayer2()

returns the second connection layer for this contact

getLayers()

returns the lists of routing, interconnect and additional layers for this contact

getNumCuts()

returns the number of cuts for this contact

getNumHVCuts()

returns the number of horizontal and vertical cuts for this contact

getRefBox()

returns the reference box for this contact

setMinCuts(unsigned int *minCuts*)

specifies minimum number of cuts to be generated for this Contact object; note that the contact size will automatically be adjusted to accommodate this number of cuts.

Feedback

stretch(Box *refBox*)

grow or shrink this contact until length and width match specified box dimensions

stretchTo(Direction *dir*, Point *point*)

grow or shrink this contact in specified direction until point is collinear with side

stretchToCoord(Direction *dir*, Coord *value*)

grow or shrink this contact, by defining reference box in specified direction

---

AbutContact

The AbutContact class provides a specialized contact, which is used to abut two contacts. This AbutContact class is directly derived from the Contact class. In addition to the methods inherited from the base Contact class, an additional method is provided to control the via spacing between the two abutting contacts.

AbutContact(Layer *layer1*, Layer *layer2*, NodeSpec *nodeSpec*=None, Direction *routeDir1*=None, Direction *routeDir2*=None, Point *point1*=INVALID, Point *point2*=INVALID, Direction *anchor*=Direction.CENTER, LayerList *addLayers*=None, string *name*='', Direction *abutDir*=Direction.NONE, int *abutViaSpaceFactor*=1, bool *symAddLayer*=False)

creates an AbutContact object which can be used to abut two contacts. The *abutDir* specifies the direction in which this contact should be abutted with another contact.

clone()

returns cloned copy of this AbutContact object

setAbutViaSpaceFactor(int *abutViaSpaceFactor*)

sets via spacing factor as integer multiple of one half via spacing design rule value

ArrayInstContact

The ArrayInstContact class provides a specialized contact, which only differs from the base Contact class as regards the implementation of cuts. The cuts for this contact are implemented using the InstanceArray class, rather than as separate rectangles. This is more memory efficient for large contacts, which require a larger number of cuts.

ArrayInstContact(Layer *layer1*, Layer *layer2*, NodeSpec *nodeSpec*=None, Direction *routeDir1*=None, Direction *routeDir2*=None, Point *point1*=INVALID, Point

Feedback

point2=INVALID, Direction *anchor*=CENTER, LayerList *addLayers*=None, string *name*='')

creates an ArrayInstContact object; note that this creation method has exactly the same parameters as the base Contact class.

clone()

returns cloned copy of this ArrayInstContact object

Grouping

The Grouping class provides the ability to group together one or more physical components into a logical grouping. The methods for this class provide convenient layout manipulation operations, which can be applied to all components in a single operation.

Grouping(string *name*='', *components*=None)

creates Grouping object with given name; *components* parameter specifies physical component or list of components which should be added to this new grouping.

add(*components*)

add physical component or list of physical components to this Grouping object

clone()

returns a cloned copy of this Grouping object

destroy()

base class method which destroys container object for this Grouping object

find(string *name*="")

returns Grouping object having this *name* in the current DloGen design

flatten()

recursively flattens this Grouping object to lowest-level physical components

getBBox(ShapeFilter *filter*)

returns bounding box for this Grouping object, using any specified layers

getComp(int *index*)

returns component in this grouping at the specified *index* value

getComps()

returns uniform list of all members for this Grouping object

getLeafComps()

returns uniform list of all leaf-level physical components in this grouping

getName()

returns name for this Grouping object

isPersistent()

returns True if this Grouping object is persistent and False otherwise

makePersistent(bool *persistent*)

makes this Grouping object persistent in the OpenAccess database

mirrorX(Coord *yCoord*=0)

mirrors all physical components about X-coordinate axis

mirrorY(Coord *xCoord*=0)

mirrors all physical components about Y-coordinate axis

moveBy(Coord *dx*, Coord *dy*)

moves all physical components *dx* units in x-direction and *dy* units in y-direction

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter())

moves all physical components in the specified direction, so that specified point becomes the handle point for the bounding box of this Grouping object.

moveTowards(Direction *dir*, Coord *distance*)

moves all physical components by specified distance in the specified direction

remove(*components*)

remove physical component or list of components from this grouping

rotate90(Point *origin*=None)

rotates all physical components by 90 degrees, in counter-clockwise direction

rotate180(Point *origin*=None)

rotates all physical components by 180 degrees, in counter-clockwise direction

rotate270(Point *origin*=None)

rotates all physical components by 270 degrees, in counter-clockwise direction

setName(string *name*)

sets the name for this grouping

Feedback

transform(Transform *trans*)

applies the transform *trans* passed as a parameter to all physical components

ungroup(Grouping *owner*=None, bool *all*=False)

transfers all grouping members to *owner* grouping (or current design, if *owner* is None), and then destroys this grouping; if all is True, all intermediate-level groupings within this grouping are also ungrouped.

Attribute:

- **name** – returns the name for this Grouping object (settable attribute)

Compound Component

The CompoundComponent class is derived from the Grouping class, and provides the ability to group together one or more physical components into a compound object. This class provides a locking mechanism for membership and member operations. Members can only be added or removed or modified when the compound component is unlocked.

CompoundComponent(string *name*, *components*=None)

creates compound component object with given name; *components* parameter specifies components which should be added to this new compound component.

clone()

returns a cloned copy of this CompoundComponent object

destroy()

destroys the CompoundComponent, as well as all of its members

flatten()

flattens this CompoundComponent to lowest-level physical components

isLocked()

returns True if this CompoundComponent is locked and False otherwise

lock()

locks this CompoundComnponent, so that members cannot be modified

mirrorX(Coord *yCoord*=0)

temporarily unlocks this CompoundComponent, and then mirrors all physical components about X-coordinate axis

Feedback

mirrorY(Coord *xCoord*=0)

temporarily unlocks this CompoundComponent, and then mirrors all physical components about Y-coordinate axis

moveBy(Coord *dx*, Coord *dy*)

temporarily unlocks this CompoundComponent, and then moves all physical components *dx* units in x-direction and *dy* units in y-direction

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter())

temporarily unlocks this CompoundComponent, and then moves all physical components so that specified point *destination* then becomes the *handle* point for the bounding box of this Grouping object.

moveTowards(Direction *dir*, Coord *distance*)

temporarily unlocks this CompoundComponent, and then moves all physical components by specified distance in the specified direction

rotate90(Point *origin*=None)

temporarily unlocks this CompoundComponent, and then rotates all physical components by 90 degrees, in counter-clockwise direction

rotate180(Point *origin*=None)

temporarily unlocks this CompoundComponent, and then rotates all physical components by 180 degrees, in counter-clockwise direction

rotate270(Point *origin*=None)

temporarily unlocks this CompoundComponent, and then rotates all physical components by 270 degrees, in counter-clockwise direction

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

temporarily unlocks this CompoundComponent, and then snaps to grid points

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

temporarily unlocks this CompoundComponent, and then snaps X-coordinate

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

temporarily unlocks this CompoundComponent, and then snaps Y-coordinate

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter())

temporarily unlocks this CompoundComponent, and then snaps in the *dir* direction

Feedback

transform(Transform *trans*)

temporarily unlocks this CompoundComponent, and then applies *trans* transform to all physical components

ungroup()

destroys CompoundComponent, and returns members to the DloGen design object

unlock()

unlocks this CompoundComponent, so that members can be modified

---

Bar

The Bar class provides simple routing on a single layer or between two different layers. This is a horizontal or vertical route, to which vertical or horizontal connections can be made. Contacts can be added to make connections between different layers.

Bar(Layer *layer*, Direction *dir*, string *node*, Point *point1*=INVALID, Point *point2*=INVALID, Direction *anchor*=SOUTH_WEST, string *name*)

creates Bar object on the specified layer, using the specified anchor direction

addContact(Layer *layer*, Point *point1*, Point *point2*=INVALID, Direction *anchor*=CENTER, Direction *routeDir*=NONE)

adds a new contact to this Bar object, using specified points to define reference box

clearContacts()

removes all of the contacts associated with this Bar object

clone()

returns a cloned copy of this Bar object

destroy()

destroys all of the components (contacts and rectangle) for this Bar object

extendTo(Point *point*)

extends this Bar until end of Bar is collinear with specified point

getContacts()

returns uniform list of all contacts associated with this Bar object

getDirection()

returns direction for this Bar object (either NORTH_SOUTH or EAST_WEST)

Feedback

getLayer()

returns layer on which this Bar object is defined

getRect()

returns Route rectangle for this Bar object

getRoutePathIntersectBox(RouteTarget *fromTarg*)

returns box which is intersection of this Bar with generate RoutePath object

stretchTo(Point *point*)

stretches (or shrinks) this Bar until end of Bar is collinear with specified point

stretchToCoord(Direction *dir*, Coord *value*)

stretches Bar in the *dir* direction, until edge of Bar has specified coordinate *value*

trim()

trims Bar object to size required to connect all associated contacts

---

ContactRing

The ContactRing class creates a guard ring, which can be used to isolate a set of devices in a design from external noise and/or to prevent latch-up. This guard ring consists of four abutting contacts, along with an optional fill rectangle. An optional gap between the top and left contacts can also be generated, when the contact ring is created.

ContactRing(Layer *layer1*, Layer *layer2*, string *node*='', LayerList *addLayers*=None, Coord *width*=0, Coord *gap*=0, LayerList *fillLayers*=None, String name='', DirectionList *locations*=None, ShapeFilter *ruleFilter*=ShapeFilter(), Box *encloseBox*=None, PhysicalComponent *encloseComp*=None, bool *overlapContact*=False, bool *fillToBoundary*=False, dict *options*=None)

creates ContactRing object using *layer* and *width* parameters to define four contacts which make up this ContactRing object. Note that this contact ring can be automatically placed to enclose all shapes and instances in the current design, or can be explicitly placed using the *encloseBox* or *encloseComp* parameters.

chop(Box *cutBox*, LayerList *cutLayers*, LayerList *mendLayers*=None,

Layer *joinLayer*=None)cuts out interconnect layers specified by *cutLayers* parameter, for the section of contact ring defined by *cutBox* parameter. The optional *mendLayers* and *joinLayer* parameters are used to restore connectivity after this chopping operation.

Feedback

clone()

returns a cloned copy of this ContactRing object

destroy()

destroys this ContactRing object, including all contacts and fill rectangle

getContact(Direction *dir*)

returns the contact for the specified direction (NORTH, SOUTH, EAST, WEST)

DeviceContact

The DeviceContact class provides a connection between adjacent interconnect layers. The DeviceContact class is directly derived from the CompoundComponent class, and is used for device construction (versus the Contact class, which is best suited for routing). This DeviceCntact can be constructed to meet DRC and electrical design rules, or the designer can easily override these design rule values. This approach provides flexibility.

DeviceContact(Layer *layer1*, Layer *layer2*, Box *box*, GapStyle *gapStyle*=GapStyle.MIN_CENTER, ulist[int] *minCuts*=None, ulist[float] *layer1Ext*=None, ulist[float] layer2Ext=None, ulist[float] viaSpace=None, ulist[float] viaSize=None, Direction *routeDir1*=None, Direction *routeDir2*=None, string *name*=”)

creates a DeviceContact object which connects the two specified layers. The *box* parameter specifies the area to be filled by the *layer1* rectangle. The *layer1Ext*, *layer2Ext*, *viaSpace* and *viaSize* parameters can be used to override design rule values. Otherwise, applicable design rules are used to construct this DeviceContact.

clone()

returns a cloned copy of this DeviceContact object

getLayer1()

returns the first connection layer for this device contact

getLayer2()

returns the second connection layer for this device contact

getNumHVCuts()

returns the number of horizontal and vertical cuts for this device contact

getRect1()

returns the rectangle defined for the first interconnect layer

Feedback

getRect2()

returns the rectangle defined for the second interconnect layer

getViaBBox()

returns bounding box of the via cuts for this device contact

getViaLayer()

returns via layer for this device contact

stretch(Layer *layer*, Box *box*)

stretches rectangle on *layer* until length and width match specified box dimensions

stretchToCoord(Layer *layer*, Direction *dir*, Coord *value*)

stretches rectangle on *layer* in specified direction *dir* until aligned with *value*

stretchToPoint(Layer *layer*, Direction *dir*, Point *point*)

stretches rectangle on *layer* in specified direction *dir* until aligned with *point*

MultiPath

The MultiPath class provides the ability to group together multiple path objects. Additional subpath objects can be generated based upon the original master path object.

MultiPath(Layer *layer*, PointList *points*, Coord *width*, Direction *justify*=EAST_WEST, Coord *sep*=0, PathStyle *style*=PathStyle.TRUNCATE, Coord *beginExt*=0, Coord *endExt*=0, bool *choppable*=True, ulist[Box] *chopBoxes*=None, string *node*="", string *masterName*="", string *name*= "")

creates MultiPath object, using the specified values to create a Path object, based upon the master path, which is added to this newly created MultiPath object.

clone()

returns a cloned copy of this MultiPath object

createEnclosureSubpath(Layer *layer*, Coord *enclosure=0,* Coord *beginEncl*=None, Coord *endEncl*=None, bool *choppable*=True, string *node*="", string *name*="")

adds a new Path object to this MultiPath, defined by specified enclosure values

Feedback

createOffsetSubpath(Layer *layer*, Coord *width*, Direction *justify*=EAST_WEST, Coord *sep*=0, Coord *beginOffset*=0, Coord *endOffset*=0, bool *choppable*=True, string *node*="", string *name*="")

adds a new Path object to this MultiPath, defined by specified justification and separation values

createSubrectangles(Layer *layer*, Coord *width*=0, Coord *length*=0, Coord *space*=0, GapStyle *gapStyle*=MINIMUM, Direction *justify*=EAST_WEST, Coord *sep*=0, Coord *beginOffset*=None, Coord *endOffset*=None, bool *choppable*=True, string *node*="" string *name*="", bool *diagonal*=False, PlaceStyle *parMainPath*=PlaceStyle.WIDTH_PARALLEL_X_AXIS, *fillSchema*=False, bool *distSingleRect*=True, int *layerGridSizeFactor*=2)

creates a field of subrectangles on the specified *layer* for this MultiPath

destroy()

destroys this MultiPath object, including all path rectangles

genJustifyPathPoints (PointList *refPoints*, Coord *width*=0, Direction *justify*= EAST_WEST, Coord *sep*=0, Coord *subWidth*=0, Coord *beginOffset*=0, Coord *endOffset*=0)

returns justified point list constructed from *refPoints* list of reference points

getChoppedSubpathPointLists(*name*="")

returns list of points for chopped subpath or master path specified by *name*

getMasterPathName()

returns name assigned to master path for this MultiPath

getSubpathPoints(*name*="")

returns list of points for subpath or master path specified by *name*

mirrorX(Coord *yCoord* = 0)

mirrors all components in this MultiPath about the X coordinate axis

mirrorY(Coord *xCoord* = 0)

mirrors all components in this MultiPath about the Y coordinate axis

moveBy(Coord *dx*, Coord *dy*)

moves all components *dx* units in x-direction and *dy* units in y-direction

Feedback

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter())

moves all components so that specified point *destination* becomes the *handle* point for the bounding box of this MultiPath object

moveTowards(Direction *dir*, Coord *distance*)

moves all components by specified distance in the specified direction

rotate90(Point *origin*=None)

rotates all components by 90 degrees in a counter-clockwise direction

rotate180(Point *origin*=None)

rotates all components by 180 degrees in a counter-clockwise direction

rotate270(Point *origin*=None)

rotates all components by 270 degrees in a counter-clockwise direction

setChopBoxes(BoxList *chopBoxes*)

chops the Master Path by the *chopBoxes* parameter

setChoppable(bool *choppable*)

specifies whether the Master Path can be chopped by the *choppable* parameter

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

snaps both X and Y coordinates of the lower-left vertex point of the bounding box for this MultiPath object to the grid points defined by the *grid* parameter.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

snaps only X coordinate of the lower-left vertex point of the bounding box for this MultiPath object to grid points defined by the *grid* parameter.

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter())

snaps only Y coordinate of the lower-left vertex point of the bounding box for this MultiPath object to grid points defined by the *grid* parameter.

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter())

snaps coordinates of the lower-left vertex point of the bounding box for this MultiPath object to the grid points defined by the *grid* parameter.

transform(Transform *trans*)

applies transform *trans* to all of the components in this MultiPath object

Feedback

RoutePath

The RoutePath class is used to make connections between two different shapes within a design. These connections include straight-line routes, LShapes ZShape and CShape routes. For the straight-line routes, connections can also be made to bars. Note that most class methods are static, so construction is not usually required.

Connect(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None,

Coord *width*=0, bool *genContact*=True, string *name*='') first attempts to generate straight-line route; if this is not successful, then ZShape route will be generated, if this is not successful, then LShape route will be used.

CShape(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Point *position*=None, Direction *dir*=EAST, Coord *width*=0, string *name*='')

generates CShape route between route targets, using preferred routing layerF**lightLine**(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, string *name*='')generates flight line between route targets on specified *layer*, or FlightLine layer

LShape(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Direction *dir*=EAST_WEST, Coord *width*=0, string *name*='')

generates LShape route between route targets, using preferred routing layer

StraightLine(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Coord *width*=0, bool *genContact*=True, string *name*='')

generates straight-line route between specified pins, using preferred routing layer

StraightLineToBar(RouteTarget *fromTarg*, Bar *bar*, Layer *layer*=None, Coord *width*=0, bool *genContact*=True, string *name*='')

generates straight-line route between pin and bar, using preferred routing layer

ZShape(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Point *position*=None, Direction *dir*=None, Coord *width*=0, string *name*='')

generates ZShape route between specified pins, using preferred routing layer

checkLayerWidth(Layer *layer*, Coord *width*)

compares *width* parameter to minimum width DRC rule for given *layer* parameter

clone()

returns cloned copy of this RoutePath object

Feedback

destroy()

destroys this RoutePath object, including all route rectangles and contacts

findAdjacentInterconnectLayer(LayerList *layers*, Layer *bestLayer*, bool *refAbove*=True)

returns closest interconnect layer to specified layer in list of layers

getLayer()

returns layer which is being as routing layer

Boundary

The base class from which Place and Route boundary objects are derived. This is an abstract base class, only used for class derivation, which cannot be directly created.

getBBox(ShapeFilter *filter*=ShapeFilter())

returns bounding box for this Boundary; note that since Boundary has no layers, any layers specified in the *filter* ShapeFilter object will be ignored.

getEdgeNames( )

returns list of names for each edge in the polygonal boundary

getNumEdges( )

returns total number of edges for the polygonal boundary

getPoints( )

returns uniform list of points which define polygonal boundary

setEdges(PointList *points*, ulist[string] *edgeNames*)

uses list of points and edge names to set edges and edge names for boundary

---

PRBoundary

PRBoundary class is derived from the base Boundary class, and provides a boundary for a design block, used for Place-and-Route operations. There can only be a single PRBoundary within the current design object (OpenAccess restriction).

PRBoundary(PointList *points*, ulist[string] *edgeNames*)

creates PRBoundary, using specified list of points to define polygonal boundary, and list of edge names to name each of the edges defined by polygonal boundary.

find()

returns single PRBoundary object for current design.

Via

The base class from which standard and custom vias are derived. This is an abstract base class, which is only used for class derivation, and cannot be directly created.

getBBox(ShapeFilter *filter*=ShapeFilter())

returns bounding box for this Via

getColorMask( )

returns coloring information for this Via for multi-patterning

getLayer1( )

returns bottom layer for this Via

getLayer2( )

returns top layer for this Via

getMaster(bool *buildPyObj*=False)

returns master design for this Via

getName( )

always returns None; only provided to override PhysicalComponent **getName**()

getNet( )

returns any net which is assigned to this Via

getOrientation( )

returns orientation for this Via

getOrigin( )

returns origin point for this Via

getTransform( )

returns any transform defined for this Via

getViaDefName( )

returns name of via definition from technology file for this Via

Feedback

setColorMask(string *anchorString*, int *layer1ColorShift*, int *layer2ColorShift*, int *cutLayerColorShift*)

sets coloring information for this Via for multi-patterning

setName(string *name*)

name is empty string; only provided to override PhysicalComponent **setName**()

setNet(Net *net*)

assigns specified *net* to this Via; if *net* is None, then Via is removed from net

setOrientation(Orientation *orient*)

sets orientation for this Via

setOrigin(Point *origin*)

sets origin point for this Via

setTransform(Transform *trans*)

sets transform to be applied to this Via

StdVia

The standard via class, which is derived from base Via class. This standard via has a fixed number of pre-defined parameters which can be changed to modify the default standard via. Standard via definition should be specified in associated technology file.

StdVia(string *viaDefName*, ViaParam *params* = None, string *node*= “”, Transform *trans*=None)

creates a standard via object, where *viaDefName* specifies name of standard via definition contained in the associated OpenAccess technology file, and *params* is a ViaParam object which contains all parameter values to be set for this standard via.

getCutLayer()

returns cut layer for this standard via

getImplantLayer1( )

returns implant layer for bottom layer (layer1) for this standard via

getImplantLayer2( )

returns implant layer for top layer (layer2) for this standard via

Feedback

getParams( )

returns ViaParam object which was used to set via parameters for this standard via

setParams(ViaParam *params*)

sets via parameters for this standard via, using values contained in *params*

CustomVia

The custom via class, which is derived from base Via class. This custom via has set of arbitrary parameters which can be changed to modify the custom via which is generated.

CustomVia(string *viaDefName*, ParamArray *params* = None, string *node*= “”, Transform *trans*=None, bool *checkParams*=False)

creates a custom via object, where *viaDefName* specifies name of custom via definition contained in the associated OpenAccess technology file, and *params* is a ParamArray object containing all parameter values to be set for this custom via.

getParams(ParamArray *params*=None, bool *all*=True)

returns ParamArray object which was used to set via parameters for this custom via

setParams(ParamArray *params*, bool *checkParams*=False)

sets via parameters for this custom via, using values contained in *params*

ViaDef

The base class from which standard and custom via definitions are derived. These classes represent via definitions present in the technology library which are used for creation of StdVia and CustomVia objects. Objects of these classes are never created directly in the API but returned by getStdViaDef and getCustomViaDef methods of Tech object.

getName( )

returns the name of this via definition

getLayer1( )

returns the first layer associated with this via definition

getLayer2( )

returns the second layer associated with this via definition

Feedback

StdViaDef

The standard via definition class, which is derived from base ViaDef class, represents the standard via definition existing in the technology library.

getImplantLayer1( )

returns the first implant layer associated with this standard via definition. If there is no implant layer, then None is returned.

getImplantLayer2( )

returns the second implant layer associated with this standard via definition. If there is no implant layer, then None is returned.

getParams( )

returns ViaParam object representing default parameters associated with this standard via definition.

CustomViaDef

The custom via definition class, which is derived from base ViaDef class, represents the custom via definition existing in the technology library.

getLibName()

returns the library name of the master design referenced by this custom via definition

getCellName()

returns the cell name of the master design referenced by this custom via definition

getViewName()

returns the view name of the master design referenced by this custom via definition

getParams()

returns ParamArray object representing default parameters associated with this custom via definition

Shape

The base class from which all shape objects are derived. This is an abstract base class, which is only used for class derivation, and cannot be directly created.

Feedback

getBBox()

returns bounding box for this Shape, using any specified layers

getColorMask()

returns coloring information for this Shape for double and triple patterning

getLayer()

returns the current Layer for this Shape

getName()

returns any optional name which has been assigned to this Shape

getNet()

returns any net to which this Shape has been assigned

getPin()

returns any pin to which this Shape has been assigned

setColorMask(string *anchor*, string *color*)

sets coloring information for this Shape for double and triple patterning

setLayer(Layer *layer*)

sets the current Layer for this Shape

setName(string *name*)

sets the optional name for this Shape

setNet(Net *net*)

sets the net for this Shape, so that this Shape is assigned to a net in the design; if *net* parameter is None, then this Shape is removed from any assigned net.

Attributes:

- **bbox** – returns the bounding box for this Shape
- **layer** – returns the Layer associated with this Shape (settable attribute)
- **name** – returns any optional name for this Shape (settable attribute)

Shape Classes

These basic shape classes are all derived from the base Shape class. Shape objects are drawn on a single layer, which is passed as the first parameter to the construction method. These basic shape classes include the Arc, Donut, Dot, Ellipse, Line, PathSeg,

Feedback

Polygon, Rect, SimplePath and Text classes. Note that methods are inherited from the base PhysicalComponent and Shape classes.

Arc

Arc(Layer *layer*, Box *box*, double *startAngle*=0.0, double *endAngle*=0.0, Box *arcBox*=None)

creates Arc object defined by either bounding box and start and end angles, or by calculating overlap or interception of *box* and *arcBox*

clone()

returns cloned copy of this Arc object

getEllipseBBox()

returns bounding box for ellipse which defines this arc

getStartAngle()

returns start angle for this arc

getStopAngle()

returns stop angle for this arc

setEllipseBBox(Box *box*)

sets the bounding box for ellipse which defines this arc

setStartAngle(double *startAngle*)

sets the start angle for this arc

setStopAngle(double *stopAngle*)

sets the stop angle for this arc

Donut

Donut(Layer *layer*, Point *center*, Coord *radius*, Coord *holeRadius*, double *startAngle*=0.0, double *endAngle*=360.0)

creates Donut object defined by center point, radius and hole radius, and start and end angles

clone()

returns cloned copy of this Donut object

getCenter()

returns center point for this donut

Feedback

getEndAngle()

returns the end angle for this donut

getHoleBBox()

returns bounding box for hole defined by center point and hole radius

getHoleRadius()

returns the hole radius (inner radius) for this donut

getRadius()

returns the radius (outer radius) for this donut

getStartAngle()

returns the start angle for this donut

setAngles(double *startAngle*, double *endAngle*)

sets the start and end angles for this donut

setCenter(Point *center*)

sets the center point for this donut

setEndAngle(double *endAngle*)

sets the end angle for this donut

setHoleRadius(Coord *holeRadius*)

sets the hole radius (inner radius) for this donut

setRadius(Coord *radius*)

sets the radius (outer radius) for this donut

setStartAngle(double *startAngle*)

sets the start angle for this donut

Dot

Dot(Layer *layer*, Point *origin*, Coord *width*=0, Coord *height*=0)

creates Dot object defined by origin point, along with optional width and height

clone()

returns cloned copy of this Dot object

getHeight()

returns height of this dot

Feedback

getOrigin()

returns origin point for this dot

getWidth()

returns width of this dot

setHeight(Coord *height*)

sets height of this dot

setOrigin(Point *origin*)

sets origin point for this dot

setWidth(Coord *width*)

sets width of this dot

Ellipse

Ellipse(Layer *layer*, Box *box*, double *startAngle*=0.0, double *endAngle*=360.0)

creates Ellipse object defined by its bounding box and start and end angles

clone()

returns cloned copy of this Ellipse object

genPolygonPoints()

generates list of points to approximate ellipse by a polygon

getEndAngle()

returns the end angle for this ellipse

getStartAngle()

returns the start angle for this ellipse

setAngles(double *startAngle*, double *endAngle*)

sets the start and end angles for this ellipse

setBBox(Box *box*)

sets the bounding box for this ellipse

setEndAngle(double *endAngle*)

sets the end angle for this ellipse

setStartAngle(double *startAngle*)

sets the start angle for this ellipse

Feedback

Line

Line(Layer *layer*, PointList *points*)

creates Line object defined by list of points, which should not contain any coincident (duplicate) or collinear points

clone()

returns cloned copy of this Line object

getNumPoints()

returns the number of points contained in this line

getPoints()

returns the list of points which defines this line

setPoints(PointList *points*)

sets the list of points which defines this line

Attribute:

- **points** – returns or sets the list of points which defines this line

Path

Path(Layer *layer*, Coord *width*, PointList *points*, PathStyle *style*=PathStyle.TRUNCATE)

creates Path object defined by list of points, which should not contain any coincident (duplicate) or collinear points (use PointList **compress**() method)

clone()

returns cloned copy of this Path object

getBeginExt()

gets begin extension value for this path; only valid for VARIABLE path styles

getBoundary(bool *usePathOrder*=False)

returns list of points which defines the boundary for this path

getEdgeLength(int *index*)

returns the length from the Path object for the specified segment number

getEndExt()

returns end extension value for this path; only valid for VARIABLE path styles

Feedback

getNumPoints()

returns the number of points which defines this path

getPoints()

returns the list of points which defines this path

getStyle()

returns path style, as TRUNCATE, EXTEND, ROUND or VARIABLE

getWidth()

returns the width for this path

isOrthogonal()

returns True if all points in this path are orthogonal, otherwise False

setBeginExt(Coord *beginExt*)

sets begin extension value for this path; only valid for VARIABLE path styles

setEndExt(Coord *endExt*)

sets end extension value for this path; only valid for VARIABLE path styles

setPoints(PointList *points*)

sets the list of points which defines this path

setStyle(string *style*)

sets the path style, as TRUNCATE, EXTEND, ROUND or VARIABLE

setWidth(Coord *width*)

sets the width for this path

PathSeg

PathSeg(Layer *layer*, Point *begin*, Point *end*)

creates PathSeg path segment object, defined by begin and end points; these two points must define horizontal, vertical or diagonal segment, or exception is raised.

clone()

returns cloned copy of this PathSeg object

getBoundary()

returns list of points which define the boundary for this path segment

Feedback

getPoints()

returns the begin point and end point for this path segment

getWidth() returns the width for this path segment

isOrthogonal()

returns True if the points for this path segment are orthogonal, otherwise False

setPoints(Point *beginPoint*, Point *endPoint*)

sets the begin point and end point for this path segment

setWidth(Coord *width*)

sets the width for this path segment

Polygon

Polygon(Layer *layer*, PointList *points*, bool *compress*=False)

creates Polygon object defined by list of points, which should not contain any coincident (duplicate) or collinear points (*compress* option removes such points)

clone()

returns cloned copy of this Polygon object

getEdgeLength(int *index*)

returns the length from the Polygon object for the specified segment number

getNumPoints()

returns number of points which define this polygon

getPoints()

returns the list of points which defines this polygon

isOrthogonal()

returns True if all vertex points in this polygon are orthogonal, otherwise False

setPoints(PointList *points*)

sets the list of points which defines this polygon

subdivide(unsigned *numPoints*, bool *verticalChop*=False)

subdivides this polygon into sub-polygons, having fewer than *numPoints* points; *verticalChop* specifies whether to subdivide the polygon object in the horizontal (False) or vertical (True) direction

Feedback

Attribute:

- **points** – returns the list of points which defines this polygon

Rect

Rect(Layer *layer*, Box *box*)

creates Rect rectangle object defined by bounding box

clone()

returns cloned copy of this Rect object

expand(Coord *coord*)

expands this rectangle by coordinate value in each direction

expandDir(Direction *dir*, Coord *coord*)

expands this rectangle by *coord* coordinate value in specified *dir* direction

expandToGrid(Grid *grid*, Direction *dir*=None)

expands this rectangle in direction *dir* to align with nearest grid point

fillBBoxWithRects(Layer *layer*, Box *box*, Coord *width*=None, Coord *height*=None, Coord *spaceX*=None, Coord *spaceY*=None, GapStyle *gapStyle*=GapStyle.MINIMUM, Grouping *group*=None)

fills specified *box* with equally sized rectangles on *layer*

fillDiagBoxWithRects(Layer *layer*, Box *diagBox*, Coord *width*=None, Coord *height*=None, Coord *space45*=None, Coord *space135*=None, GapStyle *gapStyle*=GapStyle.MINIMUM, Grouping *group*=None)

fills specified diagonal *box* with equally sized rectangles on *layer*

fillDiagBoxWithDiagRects(Layer *layer*, Box *diagBox*, Coord *width45*, Coord *height135*, Coord *space45*, Coord *space135*, GapStyle *gapStyle*=GapStyle.MINIMUM, Grouping *group*=None)

fills specified diagonal *box* with equally sized diagonal rectangles on *layer*

getBottom()

returns coordinate value for bottom of this rectangle

getCoord(Direction *dir*)

returns coordinate of this rectangle in the *dir* direction

getHeight()

returns height of this rectangle

Feedback

getLeft()

returns coordinate value for left side of this rectangle

getRight()

returns coordinate value for right side of this rectangle

getTop()

returns coordinate value for top of this rectangle

getWidth()

returns width of this rectangle

setBBox(Box *box*)

sets the bounding box which defines this rectangle

setBottom(Coord *v*)

sets coordinate value for bottom of this rectangle

setCoord(Direction *dir*, Coord *coord*)

sets this rectangle to have coordinate value in the *dir* direction.

setLeft(Coord *v*)

sets coordinate value for left side of this rectangle

setRight(Coord *v*)

sets coordinate value for right side of this rectangle

setTop(Coord *v*)

sets coordinate value for top of this rectangle

Attributes:

- **bbox** – returns the bounding box which defines this rectangle
- **bottom** – returns coordinate value for bottom of rectangle (settable attribute)
- **left** – returns coordinate value for left side of rectangle (settable attribute)
- **right** – returns coordinate value for right side of rectangle (settable attribute)
- **top** – returns coordinate value for top of rectangle (settable attribute)

Feedback

Text

Text(Layer *layer*, string *text*, Point *origin*, Coord *height*)

creates Text object defined by text string, placed at origin point, using *height*

clone()

returns cloned copy of this Text object

getAlignment()

returns the horizontal and vertical alignment location for this Text object

getFont()

returns the font for this Text object

getHeight()

returns the height for this Text object

getOrientation()

returns the orientation for this Text object

getOrigin()

returns the origin point for this Text object

getText()

returns the text string for this Text object

hasOverbar()

returns True if an overbar is displayed, otherwise False

isDrafting()

returns True if drafting mode is enabled, otherwise False

isVisible()

returns True if this Text object is visible, otherwise False

setAlignment(Location *location*)

sets the horizontal and vertical alignment location for this Text object

setDrafting(bool *drafting*)

sets Boolean flag to control drafting mode for this Text object

setFont(Font *font*)

sets the font for this Text object

Feedback

setHeight(Coord *height*)

sets the height for this Text object

setOrientation(Orientation *orient*)

sets the orientation for this Text object

setOrigin(Point *origin*)

sets the origin point for this Text object

setOverbar(bool *overbar*)

sets Boolean flag to control display of an overbar for this Text object

setText(string *text*)

sets the text string for this Text object

setVisible(bool *visible*)

sets Boolean flag to control visibility of this Text object

TextDisplay

The abstract base class from which attribute display objects are derived. Note that this class is not directly constructed; all class methods will be inherited by AttrDisplay class.

getAlignment()

returns the horizontal and vertical alignment location for this TextDisplay object

getFont()

returns the font for this TextDisplay object

getFormat()

returns the text display format for this TextDisplay object

getHeight()

returns the height for this TextDisplay object

getOrientation()

returns the orientation for this TextDisplay object

getOrigin()

returns the origin point for this TextDisplay object

getText()

returns the text string for this TextDisplay object

Feedback

hasOverbar()

returns True if an overbar is displayed, otherwise False

isDrafting()

returns True if drafting mode is enabled, otherwise False

isVisible()

returns True if this Text object is visible, otherwise False

setAlignment(Location *location*)

sets the horizontal and vertical alignment location for this TextDisplay object

setDrafting(bool *drafting*)

sets Boolean flag to control drafting mode for this TextDisplay object

setFont(Font *font*)

sets the font for this TextDisplay object

setFormat(TextDisplay.Format *format*)

sets the text display format for this TextDisplay object

setHeight(Coord *height*)

sets the height for this TextDisplay object

setOrientation(Orientation *orient*)

sets the orientation for this TextDisplay object

setOrigin(Point *origin*)

sets the origin point for this TextDisplay object

setOverbar(bool *overbar*)

sets Boolean flag to control display of an overbar for this TextDisplay object

setVisible(bool *visible*)

sets Boolean flag to control visibility of this TextDisplay object

AttrDisplay

The derived class from which attribute display objects are created; note that this class is always derived from the TextDisplay abstract base class.

AttrDisplay(*obj*, *attribute*, Layer *layer*, Point *origin*, Coord *height*, Location *location*=Location.UPPER_LEFT, Orientation *orient*=Orientation.R0, Font

Feedback

font=Font.STICK, TextDisplay.Format *format*=TextDisplay.Format.NAME_VALUE, bool *overbar*=False, bool *visible*=True, bool *drafting*=False)

creates attribute display for specified *obj* design object , with specified *attribute* attribute type. The *obj* object is design object corresponding to one of DloGen, Instance, InstTerm, Net or Term classes. The specified *attribute* uses one of pre-defined enumerated classes DloGen.AttrType, Instance.AttrType, InstTerm.AttrType, Net.AttrType or Term.AttrType. Note that attribute type should correspond to design object type.

getAttribute()

returns display attribute type for this attribute display

getObject()

returns design object for this attribute display

Symbolic constant values used by attribute displays are defined as follows:

Class Name	Attributes
TextDisplay.Format	NAME, NAME_VALUE, VALUE
DloGen.AttrType	CELL_NAME, CELL_TYPE, LAST_SAVED_TIME, LIB_NAME, VIEW_NAME
Instance.AttrType	CELL_NAME, IS_BOUND, LIB_NAME, NAME, NUM_BITS, VIEW_NAME
InstTerm.AttrType	NAME
Net.AttrType	IS_EMPTY, IS_GLOBAL, IS_IMPLICIT, NAME, NUM_BITS, SIG_TYPE
Term.AttrType	HAS_PINS, NAME, NUM_BITS

These symbolic constant values are specified using the class name and attribute as follows: TextDisplay.Format.NAME or Term.AttrType.HAS_PINS.

Reference Classes

These basic reference classes are all derived from the base PhysicalComponent class. Reference objects are used to refer to a physical component contained in a lower-level of the design hierarchy in a hierarchical layout design. The designer can use these reference objects to directly perform geometric operations; for example, a physical component can be aligned with a poly gate rectangle which is contained within an instance of a transistor. All reference classes are derived from the base PhysicalCompRef reference class.

Feedback

PhysicalCompRef

The base class from which all reference objects are derived. Note that all reference objects are created and managed by the DloGen design class; the DloGen class method **findCompRef**() is used to create and access these reference objects.

abut(Direction *dir*, IPhysicalComponent *refComp*, ShapeFilter *filter* = ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None)

abuts component referred to by this reference component

alignEdge(Direction *dir*, IPhysicalComponent *refComp*, Direction *refDir*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Coord *offset*=None)

aligns edge of component referred to by this reference component

alignEdgeToCoord(Direction *dir*, Coord *coord*, ShapeFilter *filter*=ShapeFilter())

aligns edge of component referred to by this reference component to *coord*

alignEdgeToPoint(Direction *dir*, Point *point*, ShapeFilter *filter*=ShapeFilter())

aligns edge of component referred to by this reference component to *point*

alignLocation(Location *loc*, IPhysicalComponent *refComp*, Location *refLoc*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Point *offset*=None)

aligns location of component referred to by this reference component

alignLocationToPoint(Location *loc*, Point *point*, ShapeFilter *filter*=ShapeFilter())

aligns location of component referred to by this reference component to point

getBBox()

returns bounding box for this reference component, relative to current design

getLocationPoint(Location *loc*, Point *default*=None)

returns a point corresponding to the *loc* location parameter which can be one of nine standard (pre-defined) locations, or a custom location previously created by calling setCustomLocation().

getName()

returns leaf-level name for this reference component

getParentPathName()

returns full hierarchical path name to parent instance containingthis reference component within the current DloGen design.

getParentPathTransform()

returns transform of the parent Instance, identified by parent path name

Feedback

getPathName()

returns full hierarchical path name for this reference component

getProps()

returns properties associated with referenced physical component

getSpacing(Direction *dir*, IPhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter* = None)

returns spacing distance between this reference component and *refComp* physical component in the specified *dir* direction.

getTopInst()

returns top-level instance in design containing this reference component

keepRelativePosition(IPhysicalComponent *comp*, bool *keep* = True)

keeps the alignment of physical components throughout PyCell evaluation.

mirrorX(Coord *yCoord*=0) mirrors this reference component about the X axis

mirrorY(Coord *xCoord*=0)

mirrors this reference component about the Y axis

moveBy(Coord *dx*, Coord *dy*)

moveTo(Coord *x*, Coord *y*, Location *loc*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) moves this reference component, so bounding box handle aligns with *x* and *y* values

moveTowards(Direction *dir*, Coord *d*) moves this reference component *d* units in the *dir* direction

overlaps(Box *box*) returns True if this reference component overlaps *box*, and returns False otherwise

place(Direction *dir*, IPhysicalComponent *refComp*, Coord *distance*, ShapeFilter *filter*=ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None)

places this reference component *distance* units from the *refComp* component

rotate90(Point *origin*=None) rotates this reference component by 90 degrees

rotate180(Point *origin*=None) rotates this reference component by 180 degrees

rotate270(Point *origin*=None) rotates this reference component by 270 degrees

Feedback

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) snaps this reference component, so lower-left bounding box vertex lies on grid

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter()) snaps this reference component in the *dir* direction.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) snaps this reference component, so that X coordinate lies on grid

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) snaps this reference component, so that Y coordinate lies on grid

transform(Transform *trans*) applies transform *trans* to this reference component

Attributes:

- **bbox** – returns bounding box for this physical component reference object
- **props** – returns properties associated with referenced physical component

GroupingRef

The GroupingRef class is derived from the PhysicalCompRef class, and allows a Grouping object to be used as a reference component. Note that this Grouping should be persistent; this is done with the **makePersistent**() method for the Grouping class.

getCompRefs()

returns list of reference physical components within this GroupingRef object

InstanceRef

The InstanceRef class is derived from the PhysicalCompRef class, and allows an Instance object to be used as a reference component.

getCompRefs()

returns list of reference physical components within this InstanceRef object

getMaster(bool *buildPyObj*=False)

returns the master design object for this InstanceRef

getParams(ParamArray *params*=None, bool *all*=True)

returns array of explicit parameters used when this InstanceRef object was created

Feedback

InstanceArrayRef

The InstanceArrayRef class is derived from the PhysicalCompRef class, and allows an InstanceArray object to be used as a reference component.

getMemberRefs()

returns list of reference physical components which are members of this InstanceArray object

VectorInstanceRef

The VectorInstanceRef class is derived from the InstanceRef class, and is used to provide a reference to a lower-level vector instance object which is contained within the hierarchy of a hierarchical physical layout design.

getCompRefs()

returns a uniform list of all of the reference physical components which are contained inside this VectorInstanceRef object.

ShapeRef

The ShapeRef class is provided to allow the designer to easily access read-only shape information, such as is associated with connectivity. This includes information about shapes in the instance master or submaster. Note that there is not any direct OpenAccess equivalent object for this ShapeRef class object.

getBBox(ShapeFilter *filter*=ShapeFilter())

returns bounding box, using all layers specified by the *filter* ShapeFilter parameter; converts result from instance submaster coordinates to current design coordinates.

getColorMask()

returns coloring information for this Shape for double and triple patterning

getInst()

returns associated instance for this ShapeRef object

getInstPin()

returns associated instance pin for this ShapeRef object

getLayer()

returns layer on which the referenced shape has been drawn

getName()

returns the optional name for the referenced shape

Feedback

getNetRef()

returns the associated net for this ShapeRef object

getPinRef()

returns the associated pin for this ShapeRef object

getTransform()

returns Transform for associated instance of this ShapeRef object

Attributes:

- **bbox** –bounding box for this ShapeRef object
- **layer** –layer for this ShapeRef object
- **name** –name for this ShapeRef object

ArcRef

The ArcRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is an arc shape.

DonutRef

The DonutRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a donut shape.

DotRef

The DotRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a dot shape.

EllipseRef

The EllipseRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is an ellipse shape.

LineRef

The LineRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a line shape.

PathRef

The PathRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a path shape.

getBeginExt()

returns beginning extension value for the referenced path

Feedback

getBoundary(bool *usePathOrder*=False)

returns list of the points which define the referenced path

getEndExt()

returns ending extension value for the referenced path

getNumPoints()

returns the number of points for the referenced path

getPoints()

returns the list of points that define the referenced path

getStyle()

returns the end point style for the referenced path

getWidth()

returns width of the referenced path

isOrthogonal()

returns True, if all points in the referenced path are orthogonal; False otherwise.

PathSegRef

The PathSegRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a path segment shape.

getBoundary()

returns list of points which define the boundary for the referenced path segment

getPoints()

returns begin point and end point for the referenced path segment

getWidth()

returns width of the referenced path segment

isOrthogonal()

returns True, if points in referenced path segment are orthogonal; False, otherwise.

PolygonRef

The PolygonRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a polygon shape. Additional methods are provided to obtain information about the referenced polygon shape.

Feedback

getNumPoints()

returns number of points that define the referenced polygon

getPoints()

returns list of points that define the referenced polygon

isOrthogonal()

returns True, if all points in referenced polygon are orthogonal; False otherwise.

RectRef

The RectRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a rectangle shape.

TextRef

The TextRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a text shape.

getText()

returns the text string from the Text object which is being referenced by this TextRef object

TextDisplayRef

The TextDisplayRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a text display shape.

AttrDisplayRef

The AttrDisplayRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is an attribute display shape.

Basic Geometric Classes

The basic geometric classes are used to represent basic geometric objects through use of the Point and Box classes. In addition, the Direction, Location, Orientation and Transform classes are provided to work with these basic geometric objects.

Point

Basic geometric point class, used to represent a single point in the x-y coordinate space used to define layout objects. A point is defined by its x and y coordinate values.

The Python addition and subtraction operators have been overloaded to work with points.

Feedback

Point(Coord *x*=0, Coord *y*=0)

creates Point object, using the specified x and y coordinate values

areColinearPoints(Point *p1*, Point *p2*, Point *p3*)

returns True if 3 points are collinear or coincident, and returns False otherwise

copy()

returns a copy of this Point object

getCoord(Direction *dir*)

returns the coordinate value in the specified direction *dir* for this point

getSpacing(Direction *dir*, Point *refPoint*)

returns relative spacing distance from this point to *refPoint* reference point

getX()

returns x-coordinate value for this Point

getY()

returns y-coordinate value for this Point

invalid()

returns the pre-defined invalid point value in the x-y coordinate space

isBetween(Point *a*, Point *b*)

Returns True if this Point lies between specified points on either horizontal or vertical line segment; returns False otherwise. Typically used for routing purposes.

isValid()

returns True if this Point represents valid point in x-y coordinate space

place(Direction *dir*, Point *refPoint*, Coord *distance*, bool *align*=True) –

places this point *distance* units from *refPoint* reference point in given direction *dir*

set(Point *p*)

set(Coord *_x*, Coord *_y*)

sets the point value (or coordinate value) for this point

setCoord(Direction *dir*, Coord *coord*)

sets the *coord* coordinate value in the specified direction *dir* for this point

Feedback

setX(Coord *x*)

sets x-coordinate value for this Point object

setY(Coord *y*)

sets y-coordinate value for this Point object

snap(Coord *grid*, SnapType *snapType*=None)

snaps this point to nearest grid point, using the *snapType* rounding method

snapX(Coord *grid*, SnapType *snapType*=None)

snaps the x-coordinate of this point to nearest grid point, using *snapType* rounding

snapY(Coord *grid*, SnapType *snapType*=None)

snaps the y-coordinate of this point to nearest grid point, using *snapType* rounding

snapTowards(Coord *grid*, Direction *dir*)

snaps this point to nearest grid point in the specified direction *dir*

toDiagAxes()

transforms point specified in orthogonal coordinate axis values to diagonal axes

toOrthogAxes()

transforms point specified in diagonal coordinate axis values to orthogonal axes

transform(Transform *trans*)

applies the Transform to this Point object (see description of **Transform** class)

Attributes:

- **x** – returns value of x-coordinate for this point (settable attribute)
- **y –** returns value of y-coordinate for this point (settable attribute)

Range

This is the basic geometric range class which defines a one-dimensional range of values in the x-y coordinate space which is used to define layout objects. This range is defined by the two coordinate values for the range, the left and right values for the range.

Range(Coord *left*, Coord *right*)

Feedback

Range(Range *r*)

creates Range object, using the specified left and right coordinate values

alignEdge(Direction *dir*, Range *refRange*, Direction *refDir*=None,

Coord *offset*=None)moves this range to align in the given directions with the *refRange* reference range; if *offset* is specified, then it is added to distance required to align these two ranges.

alignEdgeToCoord(Direction *dir*, Coord *coord*)

moves this range to align in the given direction with the *coord* coordinate

compareTrueCenter()

compares floating-point user units with database integer units for center point

contains(Range *range*, bool *incEnds*=True)

returns True if this range contains the specified range; if *incEdges* is True, then *range* will be contained, if left and right coordinates are within this range.

containsCoord(Coord *coord*, bool *incEnds*=True)

returns True if this range contains *coord* coordinate; if *incEdges* flag True, then *coord* will be contained, if same value as range left or right coordinate.

expand(Coord *coord*)

expands this range by the *coord* coordinate value in both directions

expandDir(Direction *dir*, Coord *coord*)

expands this range by the *coord* coordinate value in the *dir* direction

findAlignEdgeOffset(int *moveEdgeLean*, int *refEdgeLean*)

adjusts edge alignment between floating-point user units and integer database units

fitSubranges(Coord *width*, Coord *space*, Coord *gridSize*, GapStyle *gapStyle*)

returns number of equally sized subranges which can fit within this range

fix()

checks for inverted range; if so, swaps left and right coordinates

getCenter()

returns midpoint value for this range

getCoord(Direction *dir*)

returns left or right coordinate value for this range, using *dir* direction

Feedback

getLeft()

returns left coordinate value for this range

getRight()

returns right coordinate value for this range

getWidth()

returns width of this range

hasNoWidth()

returns True if this range has zero width, and returns False otherwise

init()

sets this range to an inverted range with maximum values

intersect(Range *range*, Direction *dir*=None)

returns intersection of this range and the passed parameter *range,* using *dir* direction to determine which coordinate values should be changed.

isInverted()

returns True if right coordinate is less than left coordinate, and False otherwise

isNormal()

returns True if left coordinate is less than right coordinate, and False otherwise

limit(Coord *coord*)

returns coordinate found by limiting *coord* to be within this range

merge(Range *range,* Direction *dir*=None)

merge this range with the passed *range* parameter range, using *dir* direction to determine which coordinate values should be changed.

mergeCoord(Coord *coord*)

this range is merged with the passed *coord* parameter

moveBy(Coord *coord*)

moves this range by the specified *coord* offset coordinate value

overlaps(Range *range*, bool *incEnds*=True)

returns True if range has any overlap with *range*; if *incEdges* is True, then ranges overlap, if the left or right coordinate values are within this range.

Feedback

removeRegion(Range *range*)

removes sub- region of this range specified by the *range* parameter

set(Range *r,* Direction *dir*=None)

set(Coord *left*, Coord *right*)

sets both the left and right coordinate values for this range, using *dir* direction to determine which coordinate values should be changed.

setCenter(Coord *coord*)

sets midpoint value for this range

setCoord(Direction *dir*, Coord *coord*)

sets the left or right coordinate values, using specified *dir* direction

setDimension(Coord *coord*, Direction *dir*=None)

sets width for this range, using *dir* direction to determine which coordinate values should be changed.

setLeft(Coord *v*)

sets the left coordinate value for this range

setRight(Coord *v*)

sets the right coordinate value for this range

setWidth(Coord *width*)

sets width for this range, so that midpoint of range will be the same

Attributes:

- **left** – returns value for left coordinate of this range (settable attribute)
- **right** – returns value for right coordinate of this range (settable attribute)

Segment

Basic geometric segment class, used to define a line segment in the x-y coordinate space used to define layout objects. This segment is defined by two end point values, the head point and the tail point.

Segment(Point *head*, Point *tail*)

Feedback

Segment(Segment *s*)

creates Segment object, using the specified *head* and *tail* Point or *s* Segment values

addOffsets(Coord *begin*=0, Coord *end*=0)

adds *begin* offset value to the head point and *end* offset value to the tail point

contains(Segment *segment*, bool *incEnds*=True)

returns True if specified Segment is included in this Segment, and returns False otherwise. If *incEnds* is True, then end points of this Segment will be used.

containsPoint(Point *point*, bool *incEnds*=True)

returns True if specified Point is included in this Segment, and returns False otherwise. If *incEnds* is True, then end points of this Segment will be used.

extrapIntersect(Segment *segment*)

returns intersection of this Segment and specified Segment, after extrapolating each of these segments into complete lines.

genJustifySegment(Direction *justify*, Coord *sep*)

returns justified segment constructed from this Segment

getDeltaX()

returns difference in x-coordinate values between tail and head end points

getDeltaY()

returns difference in y-coordinate values between tail and head end points

getDir()

returns Direction associated with this Segment, by moving from head to tail

getHead()

returns the head end point value for this segment

getPosition(double *position*)

returns Point on this Segment determined by value of *position* parameter

getTail()

returns the tail end point value for this segment

hasIntersection(Segment *segment*, bool *incEnds*=True, bool *incParallel*=True)

returns True if Segment has intersection with *segment,* and returns False otherwise

Feedback

intersect(Segment *segment*)

returns intersection of this Segment with *segment*

isCoincident()

returns True if head and tail points are coincident, and returns False otherwise

isHorizontal ()

returns True if this Segment is horizontal, and returns False otherwise

isOrthogonal()

returns True if this Segment is orthogonal, and returns False otherwise

isParallel(Segment *segment*)

returns True if this Segment is parallel to *segment,* and returns False otherwise

isVertical ()

returns True if this Segment is vertical, and returns False otherwise

moveBy(Coord *dx*, Coord *dy*)

moves this Segment by the specified *dx* and *dy* offset coordinate values

reverse ()

reverses this Segment, by interchanging head and tail end point values

set(Point *head*, Point *tail*)

sets the head and tail end point values for this Segment

setHead(Point *headl*)

sets the head end point value for this Segment

setTail(Point *tail*)

sets the tail end point value for this Segment

transform(Transform *trans*)

applies *trans* transform to the head and tail points of this Segment

Attributes:

- **head** – returns value for head end point of this segment (settable attribute)
- **tail** – returns value for tail end point of this segment (settable attribute)

Feedback

Box

Basic geometric box class, used to specify bounding box which is used to define basic shapes for layout design objects.

Box(Coord *left*=INT_MAX, Coord *bottom*=INT_MAX, Coord *right*=INT_MIN, Coord *top*=INT_MIN)

Box(Point *lowerLeft*, Point *upperRight*)

Box(Box *b*)

creates Box object using parameter values to define the dimensions of the box

abut(Direction *dir*, Box *refBox*, bool *align*=True)

abuts edge of box to opposite edge of *refBox*, using edge specified by *dir* direction

alignEdge(Direction *dir*, Box *refBox*, Direction *refDir*=None, Point *offset*=None)

aligns an edge of this box with an edge of the specified *refBox* reference box

alignEdgeToCoord(Direction *dir*, Coord *coord*)

aligns an edge of this box to the specified coordinate, in the specified *dir* direction

alignEdgeToPoint(Direction *dir*, Point *point*)

aligns an edge of this box to the specified point, in the specified *dir* direction

alignLocation(Location *loc*, Box *refBox*, Location *refLoc*=None, Point *offset*=None)

aligns *loc* location point for this box with the *refLoc* location point for the *refBox*

alignLocationToPoint(Location *loc*, Point *pt*)

aligns *loc* location point for this box with the specified point

centerCenter()

returns center center point for this box

centerLeft()

returns center left point for this box

centerRight()

returns center right point for this box

Feedback

contains(Box *box*, bool *incEdges*=True)

returns True if this box contains the specified box; if *incEdges* is True, then *box* will be considered contained if it touches edges of this box.

containsPoint(Point *pt*, bool *incEdges*=True) returns True if this box contains the specified point; if *incEdges* is True, then point will be considered contained if it touches edges of this box.

expand(Coord *c*)

expands box by coordinate value in each direction

expandDir(Direction *dir*, Coord *coord*)

expands box by coordinate value in specified direction

expandForMinArea(Direction *dir*, AreaType *minArea*, Grid *grid*=None)

expands box in direction *dir*, so that the area is at least *minArea*; used to check that Box meets minimum area design rules.

expandForMinWidth(Direction *dir*, Coord *minWidth*, Grid *grid*=None)

expands box in direction *dir*, so that height or width is at least *minWidth*; used to check that Box meets minimum width design rules.

expandToGrid(Grid *grid*, Direction *dir*=None)

expands box in direction *dir* to align with nearest grid point

fix()

checks to see if box is inverted; if so, swaps coordinates and returns resulting box

getArea()

returns area of this box, which can be negative for an inverted box

getBottom()

returns coordinate value for bottom of this box

getCenter()

returns coordinates for center point of this box

getCenterX()

returns x-coordinate value for center point of this box

getCenterY()

returns y-coordinate value for center point of this box

Feedback

getCoord(Direction *dir*)

returns coordinate of this box in the specified direction

getDimension(Direction *dir*)

returns dimensions of this box in the specified direction

getHeight()

returns height of this box

getLeft()

returns coordinate value for left side of this box

getLocationPoint(Direction *dir*)

returns location point for this box, at given direction

getLocationPoint(Location *loc*)

returns location point for this box, at given location

getPoints()

returns list of four points which are defined by corners of this box

getRange(Direction *dir*)

returns coordinate values for this box in the specified direction, as a range value

getRangeX()

returns left and right coordinate values for this box as a range value

getRangeY()

returns bottom and top coordinate values for this box as a range value

getRight()

returns coordinate value for right side of this box

getSpacing(Direction *dir*, Box *refBox*)

returns spacing value between this box and reference box in the *dir* direction

getTop()

returns coordinate value for top of this box

getWidth()

returns width of this box

Feedback

hasNoArea()

returns True if this box has no area

init()

returns an inverted box with maximum values used for top, bottom, left and right

intersect(Box *box*)

returns intersection of this box with the specified box

intersect(Box *box*, Direction *dir*)

returns intersection of this box with the specified box in the specified direction

isInverted()

returns True, if this box is inverted

isNormal()

returns True if this box is not inverted, and returns False otherwise

limit(Point *point*)

returns point which is found by limiting *point* to be within this box

lowerCenter()

returns lower center point for this box

lowerLeft()

returns lower left point for this box

lowerRight()

returns lower right point for this box

merge(Box *box*)

merges this box with specified box, returning resulting merged box

merge(Box *box*, Direction *dir*)

merges this box with specified box in specified direction, returning merged box

mergePoint(Point *p*)

merges this box with the specified point, and returns the resulting merged box

mirrorX(Coord *yCoord*=0)

mirrors this box about the X coordinate axis

Feedback

mirrorY(Coord *xCoord*=0)

mirrors this box about the Y coordinate axis

moveBy(Coord *dx*, Coord *dy*)

moves this box by the specified *dx* and *dy* offset coordinate values

moveTo(Point *destination*, Location *loc*=Location.CENTER_CENTER)

moves this box to specified *destination* point, coinciding with *loc* location

moveTowards(Direction *dir*, Coord *d*)

moves this box by *d* coordinate value distance, in specified *dir* direction

overlaps(Box *box*, bool *incEdges*=True)

returns True if this box has any overlap with the specified box; if *incEdges* is True, then boxes will be considered to overlap if the edges touch.

place(Direction *dir*, Box *refBox*, Coord *distance*, bool *align*=True)

places this box relative to the *refBox* reference box in the specified *dir* direction;if *align* is True, then this box will first be aligned with the reference box.

removeRegion(Box *box*)

removes the region of this box specified by the *box* sub-box parameter

rotate90(Point *origin*=None)

rotates this box by 90 degrees, in a counter-clockwise direction

rotate180(Point *origin*=None)

rotates this box by 180 degrees, in a counter-clockwise direction

rotate270(Point *origin*=None)

rotates this box by 270 degrees, in a counter-clockwise direction

set(Coord *left*, Coord *bottom*, Coord *right*, Coord *top*)

set(Point *lowerLeft*, Point *upperRight*)

set(Box *b*)

sets this box to have the specified coordinate or box values

set(Box *b*, Direction *dir*=None)

sets this box to have the specified box value in specified direction

Feedback

setBottom(Coord *v*)

sets coordinate value for bottom of this box

setCenter(Point *point*)

sets center point for the center of this box

setCenterX(Coord *v*)

sets x-coordinate value for the center of this box

setCenterY(Coord *v*)

sets y-coordinate value for the center of this box

setCoord(Direction *dir*, Coord *coord*)

sets this box to have the coordinate value in the specified direction

setDimension(Coord *coord*, Direction *dir*)

sets dimensions of this box in the specified direction

setHeight(Coord *height*)

sets height of this box

setLeft(Coord *v*)

sets coordinate value for left side of this box

setLocationPoint(Location *loc*, Point *pt*)

moves this box, so that the *pt* point becomes the location point at the *loc* location

setRange(Direction *dir*, Range *range*)

sets this box to have the *range* coordinate values in the given direction

setRangeX(Range *range*)

sets left and right coordinate values for this box

setRangeY(Range *range*)

sets bottom and top coordinate values for this box

setRight(Coord *v*)

sets coordinate value for right side of this box

setTop(Coord *v*)

sets coordinate value for top of this box

Feedback

setWidth(Coord *width*)

sets width of this box

snap(Grid *grid*, SnapType *snapType*=None)

snaps lower left point of this box to nearest grid point

snapX(Grid *grid*, SnapType *snapType*=None)

snaps left coordinate of this box to nearest grid point.

snapY(Grid *grid*, SnapType *snapType*=None)

snaps bottom coordinate of this box to nearest grid point.

snapTowards(Grid *grid*, Direction *dir*)

snaps lower left point of this box to nearest grid point in the specified direction

transform(Transform *trans*)

apply specified transform to this box

upperCenter()

returns the upper center point for this box

upperLeft()

returns the upper left point for this box

upperRight()

returns the upper right point for this box

Attributes:

- **bottom** – returns coordinate value for the bottom of this box (settable attribute)
- **left** – returns coordinate value for the left side of this box (settable attribute)
- **right** – returns coordinate value for the right side of this box (settable attribute)
- **top** – returns coordinate value for the top of this box (settable attribute)

Direction

Defines one of several directions, along with methods to manipulate them. Defines symbolic constants for directions: NONE, NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST, NORTH_SOUTH, EAST_WEST, ANY, CENTER.

Feedback

containsComponent(Direction *dir*)

returns True if *dir* direction is a component of this Direction, and False otherwise.

extend()

extends this one dimensional Direction to a full-line direction

getJustifiedDir(Direction *justify*)

returns half-line direction perpendicular to this primary Direction, justified relative to specified *justify* direction, which should either be EAST or WEST.

getMembers()

returns list of all directions defined for the Direction class.

getPrimaryDirs()

returns list of one or more primary directions contained in this Direction.

is1Dimension()

returns True if this Direction is a one dimension direction, and False otherwise

isHalfLine()

returns True if this Direction is a one dimension direction, and False otherwise

isFullLine()

returns True if this Direction is a full-line direction, and False otherwise

isPrimary()

returns True if this Direction is a primary direction, and False otherwise

isXDir()

returns True if this Direction is defined by the X-axis, and False otherwise

isYDir()

returns True if this Direction is defined by the Y-axis, and False otherwise

mapXDirToYDir()

maps this Direction defined by the X-axis to a direction defined by the Y-axis

mapYDirToXDir()

maps this Direction defined by the Y-axis to a direction defined by the X-axis

mirrorX()

returns resulting direction when this Direction is mirrored about the X-axis

Feedback

mirrorY()

returns resulting direction when this Direction is mirrored about the Y-axis

opposite()

returns the opposite direction for this Direction object

perpendicular()

returns the perpendicular direction for this Direction object

rotate90()

returns resulting direction when this Direction is rotated 90 degrees

rotate180()

returns resulting direction when this Direction is rotated 180 degrees

rotate270()

returns resulting direction when this Direction is rotated 270 degrees

transform(Transform *trans*)

returns resulting direction when *trans* transform is applied to this Direction

---

Location

Specifies one of nine standard pre-defined locations for points on a bounding box. Defines symbolic constants for locations: LOWER_LEFT, CENTER_LEFT, UPPER_LEFT, LOWER_CENTER, CENTER_CENTER, UPPER_CENTER, LOWER_RIGHT, CENTER_RIGHT, UPPER_RIGHT.

For Rect and Polygon objects, the following attributes are reserved names for index locations:

- Each segment has three point locations: start*n* at the beginning, mid*n* in the middle, and end*n* at the end of the segment, where *n* is the segment number.
- The last segment has three additional locations: startLast, midLast, and endLast.

For Path objects, the following attributes are reserved names for index locations:

- Each centerline segment has three point locations: start*n* at the beginning, mid*n* in the middle, and end*n* at the end of the segment, where *n* is the segment number.
- Each boundary segment has point locations denoted as either Left or Right: startLeft*n*|startRight*n* at the beginning, midLeft*n*|midRight*n* in the middle, and endLeft*n*|endRight*n* at the end of the segment, where *n* is the segment number.

Feedback

- The start segment has the point location: startCenter.
- The last segment has the point locations: startLast, startLeftLast, startRightLast, midLast, midLeftLast, midRightLast, endLast, endLeftLast, endRightLast, endCenter, and endCenterLast.

In addition to the standard pre-defined locations, it is possible to create a named custom location by means of a call to setCustomLocation(). This custom location can be used with the alignLocation, alignLocationToPoint, and moveTo methods of PhysicalComponent and PhysicalCompRef.

Location(string *name*)

creates a custom or index location with the specified name. Index locations are created with the reserved names shown above. If the name of a standard location is provided, it does not create a new location and only returns the corresponding pre-defined location.

mirrorX()

returns resulting location when this Location is mirrored about the X-axis

mirrorY()

returns resulting location when this Location is mirrored about the Y-axis.

rotate90()

returns resulting location when this Location is rotated 90 degrees

rotate180()

returns resulting location when this Location is rotated 180 degrees

rotate270()

returns resulting location when this Location is rotated 270 degrees

transform(Transform *trans*)

returns resulting location when *trans* transform is applied to this Location

---

Orientation

Provides ability to rotate an object (counter-clockwise) or mirror it about the X or Y coordinates axes. In addition, a combination of rotation and mirroring can be defined.

concat(Orientation *other*)

returns result of applying these two Orientation operations in sequence

Feedback

getRelativeOrient(Orientation *o*)

returns Orientation required to convert this Orientation into the *o* Orientation

---

Transform

Provides two-dimensional transformations, consisting of orientation changes (rotation and mirroring) followed by a possible translation (offsets in the X and Y directions).

Transform(Coord *x*, Coord *y*, Orientation *o*=R0, double *mag*=1.0)

Transform(Point *offset*, Orientation *o*=R0, double *mag*=1.0)

creates Transform object defined by orientation (rotation), point (translation), and magnification

concat(Transform *transform*)

concatenates this Transform with passed *transform*, and returns result

invert()

inverts this Transform object and returns result

mirrorX(Coord *yCoord*=0)

returns Transform using MX mirroring orientation

mirrorY(Coord *xCoord*=0)

returns Transform using MY mirroring orientation

rotate90(Point *origin*=None)

returns Transform using R90 rotation orientation

rotate180(Point *origin*=None)

returns Transform using R90 rotation orientation

rotate270(Point *origin*=None)

returns Transform using R90 rotation orientation

Attributes:

- offset – returns offset point value for this Transform
- xOffset – returns x-coordinate value of the offset for this Transform
- yOffset – returns y-coordinate value of the offset for this Transform
- orientation – returns orientation value for this Transform

Feedback

Font

Specifies one of nine different fonts which can be used by a text object. Defines symbolic constants for fonts: EURO_STYLE, FIXED, GOTHIC, MATH, MIL_SPEC, ROMAN, SCRIPT, STICK and SWEDISH.

calcBBox(str *text,* Point *origin*,Coord *height*, Location *location*=Location.UPPER_LEFT, Orientation *orient*=Orientation.R0, bool *overbar*=False)

calculates bounding box for this Font object for specified text string

getMembers()

returns list of members for Font class

PathStyle

Specifies one of four path styles which can be used to specify the beginning and ending point styles for a path. Defines symbolic constants for path styles: TRUNCATE, EXTEND, ROUND, VARIABLE.

GapStyle

Specifies one of three types of spacing which should be used between rectangles in a field of rectangles. Defines symbolic constants for gap style spacing: DISTRIBUTE, PART_DISTRIBUTE, MINIMUM, MIN_CENTER.

Point List Geometry Operations

The point list geometric operations are used to provide points-in and points-out geometric operations in Python.

Point List Geometry Methods

poAnd(Collection *col1*, Collection *col2*)

performs a Boolean AND operation for Collection object *col1* and *col2*

poInside(Collection *col1*, Collection *col2*)

selects all polygons from Collection object *col1* that share their entire areas with polygons of Collection object *col2*

poNot(Collection *col1*, Collection *col2*)

performs a Boolean NOT operation for Collection object *col1* and *col2*

Feedback

poOr(Collection *col1*, Collection *col2*)

performs a Boolean OR operation for Collection object *col1* and *col*2

poOutside(Collection *col1*, Collection *col2*)

selects all polygons from Collection object *col1* with areas that lie completely outside all polygons of Collection object *col2*

poSize(Collection *col*, Coord *size*)

performs a Boolean SIZE operation for Collection object *col* by the specified *size* value

poStraddle(Collection *col1*, Collection *col2*)

selects the shapes in Collection object *col1* that the edges in Collection object *col2* would be across

poXor(Collection *col1*, Collection *col2*)

performs a Boolean XOR operation for Collection object *col1* and *col*2

Points-Collection

The points-collection is the container containing multiple closed points or multiple unclosed points.

Collection(PointList *points*=None, *objects*=None)

creates a Collection object

addPointLists(PointList | ulist [PointList] *points*)

adds the point(s) in this collection

addPointListsByObjects(PhysicalComponent | PhysicalCompRef | ulist PhysicalComponent] | ulist [PhysicalCompRef] *objects*)

adds the physical component(s) or physical component reference(s) in this collection

createPointListsToShapes(Layer *layer*, float *grid*=None, bool *asRectangle*=False)

returns the list of the Shape

getPointLists(*grid*=None)

returns the list of PointList which is being contained in this Collection object

Feedback

Technology Classes

The technology classes are used to access technology specific information stored in the Santana technology file. These classes include the Grid, Layer, ShapeFilter, and Tech classes.

Layer

This class is used to store the basic information concerning a layer in an integrated circuit, including the layer name, layer purpose and layer number.

Layer(string *name*)

creates Layer object corresponding to named layer in technology file

getAttrs()

returns six attributes as a string format for this layer

getGridResolution()

returns grid resolution value for this layer (in user units)

getLayerAbove()

getLayerAbove(LayerMaterial *material*)

returns layer above this layer, based on layer mask numbers

getLayerBelow()

getLayerBelow(LayerMaterial *material*)

returns layer below this layer, based on layer mask numbers

getLayerName()

returns name of this layer

getLayerNumber()

returns number of this layer

getMaterial()

returns the material for this layer

getPurposeName()

returns purpose name defined for this layer

Feedback

getPurposeNumber()

returns purpose number defined for this layer

getRoutingDir()

returns the routing direction for this layer

isAbove(Layer *layer*)

returns True if this layer is above the passed layer, based on layer mask numbers

isMaskLayer()

returns True if this layer is a mask layer, and False otherwise

Attributes:

- **name** – returns name for this layer
- **number** – returns number defined for this layer
- **purposeName**– returns purpose name defined for this layer
- **purposeNumber**– returns purpose number defined for this layer

---

ShapeFilter

This class is used to define the different layers which should be considered when making bounding box calculations as well as placement calculations. This class consists of a list of layers, along with an optional Boolean flag.

ShapeFilter()

ShapeFilter(Layer *layer*)

ShapeFilter(LayerList *layerList*)

creates ShapeFilter object, which can be empty, a single layer or list of layers

exclude(*layers*)

removes layer (or layer list) from list of layers selected by this ShapeFilter

excludeTexts()

filters out all Text and AttrDisplay objects

include(*layers*)

adds layer (or layer list) to list of layers selected by this ShapeFilter

Feedback

isIncluded(Layer *layer*)

returns True if *layer* parameter specifies layer which is selected by this ShapeFilter

---

LayerMaterial

Specifies the layer material to be used for physical layer on a chip. Defines symbolic constants for layer material: NWELL, PWELL, NDIFF, PDIFF, NIMPLANT, PIMPLANT, POLY, CUT, METAL, CONTACTLESS_METAL, DIFF, RECOGNITION or UNKNOWN.

PhysicalRule

Represents the values of a physical design rule, either as a single floating-point number, pair of floating-point numbers, or list of pairs of floating-point numbers.

It is derived from Python float class, so a single number can be used just like a float.

Attributes:

- **comment** – returns any comment of this physical design rule
- **id** – returns the id of this physical design rule
- **properties** – returns any properties defined for this physical design rule
- **value** – returns value of this physical design rule

RuleProperty

Represents properties associated with a physical design rule in the technology file;

Similar to the use of parameter names and values used for conditional design rules.

Attributes:

- **name** – name of rule property
- **value** – value of associated rule property

DeviceContext

This class is used to allow the PyCell author to automatically use different design rules when certain devices are constructed. For example, special design rules are typically used when creating high-voltage devices. If a device context is defined in the technology file for high-voltage devices, technology query functions for physical design rules will return

Feedback

values which should be used for creating high-voltage devices. In addition, FG methods such as “fgPlace()” would automatically use these high-voltage device values.

getLayers()

returns list of layers defined for this device context in the technology file

getName()

returns name assigned to this device context in the technology file

getRuleSubstitutions()

returns Python dictionary containing different rule substitutions defined for this device context in the technology file.

isEmpty()

returns true if this device context is empty, and returns False otherwise

Attributes:

- **layers** – list of layers for this device context
- **name** – name assigned to this device context
- **ruleSubstitutions** – list of rule substitutions for this device context
- **emptyContextName** – name of pre-defined empty device context

Ruleset

This class allows the PyCell author to access different rule sets defined in the Santana technology file. A design rule set is simply a named set of design rules in the technology file, which can be referenced and activated using the design rule set name in the PyCell code. For example, “recommended” design rule sets can be used in PyCell construction; these design rules can be used to improve manufacturing yield, but are not required to be used to verify basic design rule correctness for a design.

getAncestor()

returns ancestor design rule set for this design rule set

getName()

returns name assigned to this design rule set

Feedback

Attributes:

- **name** – name assigned to this design rule set
- **ancestor** – ancestor design rule set for this design rule set
- **defaultName** – name used by default design rule set

Tech

This class stores all technology specific information contained in the Santana technology file for a given process technology. This technology specific information contains process specific physical design rules, as well as electrical design rules. In addition, conditional physical design rules are supported, where a design rule value depends upon one or more parameters. Note that this Tech class object is automatically created and associated with the DloGen, Lib and ParamSpecArray class objects.

conditionalRuleExists(string *ruleName*, list *paramNames*)

conditionalRuleExists(string *ruleName*, Layer *layer1*, list *paramNames*)

conditionalRuleExists(string *ruleName*, Layer *layer1*, Layer *layer2*, list *paramNames*)

returns True if the named conditional design rule exists in the technology file

dbu2uu(int *value*, ViewType *viewType*=MASK_LAYOUT)

returns user unit value corresponding to specified database unit value

dbu2uuArea(int *value*, ViewType *viewType*=MASK_LAYOUT)

returns user unit value corresponding to specified square database unit value

deviceContextExists (string *name*)

returns True if named device context exists in technology file, and False otherwise

electricalRuleExists(string *ruleName*)

electricalRuleExists(string *ruleName*, Layer *layer1*)

electricalRuleExists(string *ruleName*, Layer *layer1*, Layer *layer2*)

returns True if the named electrical design rule exists in the technology file

get(string *techLibName*, *cnTechVersion*=None)

returns technology object corresponding to technology library name

Feedback

getActiveDeviceContext()

returns currently active device context for this Tech object

getActiveRuleset()

returns currently active rule set for this Tech object

getCustomViaDef()

returns CustomViaDef object corresponding to the via definition name.

getCustomViaDefNames()

returns a list of names of all custom via definitions existing in this technology.

getDeviceContexts()

returns list of all device contexts defined for this Tech object

getElectricalRule(string *ruleName*)

getElectricalRule(string *ruleName*, Layer *layer1*)

getElectricalRule(string *ruleName*, Layer *layer1*, Layer *layer2*)

returns the value for the named electrical design rule stored in the technology file; if this named electrical design rule does not exist, then an exception is raised.

getGridResolution()

returns default manufacturing grid resolution value (in user units)

getIntermediateLayers(Layer *layer1*, Layer *layer2*)

returns lists of route layers and cut layers defined between these two layers

getLayer(string *layerName*)

getLayer(unsigned int *layerNumber*)

getLayer(string *layerName*, string *purposeName*)

getLayer(unsigned int *layerNumber*, unsigned int *purposeNumber*)

returns the requested layer from the Santana technology file

getMaskColorNames()

returns a list of all mask color names defined in this technology

Feedback

getMosfetParams(string *type*, string *oxide*, string *parameter*)

returns value stored in Santana technology file for MOSFET transistor parameter

getOxideParams(string *oxide*, string *parameter*)

returns value stored in Santana technology file for oxide parameter

getPhysicalRule(string *ruleName*)

getPhysicalRule(string *ruleName*, Layer *layer1*)

getPhysicalRule(string *ruleName*, Layer *layer1*, Layer *layer2*)

returns the value for the named physical design rule stored in the technology file; if this named physical design rule does not exist, then an exception is raised.

getRulesets()

returns uniform list of all rule sets for this Tech object

getSantanaLayerNames()

returns list of layer names defined in the Santana technology file

getSantanaPurposeNames()

returns list of purpose names defined in the Santana technology file

getStdViaDef()

returns StdViaDef object corresponding to the via definition name.

getStdViaDefNames()

returns a list of names of all standard via definitions existing in this technology.

getTechParams(bool *local*=False)

returns a read-only Python dictionary containing technology parameters and values stored in the corresponding OpenAccess technology database.

getUserUnits(ViewType *viewType*=MASK_LAYOUT)

returns the user units for the specified view type

getValidLayers()

returns the list of all valid LPPs.

Feedback

id()

returns identification string for Santana technology file, which is a combination of the name string, revision and version numbers for the Santana technology file.

name()

returns name string for Santana technology file

physicalRuleExists(string *ruleName*)

physicalRuleExists(string *ruleName*, Layer *layer1*)

physicalRuleExists(string *ruleName*, Layer *layer1*, Layer *layer2*)

returns True if the named physical design rule exists in the technology file

revision()

returns revision number for Santana technology file

rulesetExists(string *name*)

returns True if named rule set exists in technology file, and False otherwise

uu2dbu(double *value*, ViewType *viewType*=MASK_LAYOUT)

returns database unit value corresponding to specified user unit value

uu2dbuArea(double *value*, ViewType *viewType*=MASK_LAYOUT)

returns database unit value corresponding to specified square user unit value

version()

returns version (as an unsigned integer) for Santana technology file

Attributes:

- **activeDeviceContext** – currently active device context for this Tech object
- **deviceContexts** – list of all device contexts for this Tech object
- **activeRuleset**– currently active rule set for this Tech object
- **rulesets** – list of all rule sets for this Tech object
- **techParams** – read-only Python dictionary containing technology parameters and values stored in the corresponding OpenAccess tech database.

Feedback

Connectivity Classes

The connectivity classes are used to represent basic design connectivity. These classes include the Net, Pin, Term, InstTerm, InstPin, and Topology classes.

SignalType

Defines one of several signal types which can be associated with a Net object.

Defines symbolic constants for these signal types: SIGNAL, POWER, GROUND, CLOCK, TIEOFF, TIEHI, TIELO, ANALOG, SCAN, RESET.

TermType

Defines one of several terminal types for a terminal in a design. Defines symbolic constants for these terminal types: INPUT, OUTPUT, INPUT_OUTPUT, SWITCH, JUMPER, UNUSED, TRISTATE.

Net

Represents the basic connectivity within a design. These Net objects can be connected to terminals on a design instance.

Net(string *netName*, SignalType *sigType*=SignalType.SIGNAL, bool *isGlobal*=False)

creates net in the current design, using *sigType* to specify signal type for this net; *isGlobal* Boolean flag parameter is set to True, for a global net in the design.

addShape(Shape *shapes*)

addShape(ulist[Shapes] *shapes*)

adds the shape(s) to this net; if this shape is associated with another net, thenan exception will be raised, since a shape can only be associated with a single net.

destroy()

destroys this net object, by first destroying any terminals for this net

find(*name*='')

returns the net having this name in the current DloGen design

findCreate()

returns the net having this name in the current DloGen design; if there is not any net in the current design having this name, then a new net will be created.

getBit(unsigned int *index*)

for a single-bit net, always returns this Net object.

getInstPins()

returns a list of InstPin objects associated with this net

getInstTerms()

returns a list of InstTerm objects associated with this net

getName()

returns the name for this net

getNumBits()

returns the number of bits of this Net.

getShapes()

returns list of shapes which are currently associated with this net

getSignalType()

returns the signal type value assigned to this net

getPins()

returns a list of pins which are connected to any terminal associated with this net

getTerm()

returns any terminal that is associated with this net

getVias()

returns list of all vias which are associated with this net

isGlobal()

returns a Boolean flag value indicating that this net is a global net

removeShape(Shape *shapes*)

removeShape(ulist[Shapes] *shapes*)

removes the shape(s) from this net

setGlobal()

sets the Boolean flag indicating that this net is a global net

Feedback

setName(string *name*)

sets the name for this Net object; the name of any associated terminal for this net will also be changed, ensuring that terminals and nets will have the same name.

setNetOverride (string *assignmentName*, string *netName*)

creates an oaNetConnectDef associating this single-bit net with the oaAssignment named *assignmentName*. If the oaAssignment referred to does not exist, this single-bit net is associated with the net named *netName*. An exception is thrown if the net named *netName* does not exist.

setSignalType()

sets the signal type value for this net

Attributes:

- **name** – the name of this net (settable attribute)
- **type** – the signal type for this net (settable attribute)
- **globalNet** – value of the Boolean flag for global nets (settable attribute)
- **AttrType** – display attribute type for this net

BusNet

A bus net represents a logical connection by one or more bits which are associated with common base name and vector range specification. The BusNet class is derived from the Net class. Bus nets have vector names, for example, G[1:5:2].

BusNet(string *baseName*, unsigned int *startIndex*, unsigned int *stopIndex*, unsigned int *indexStep*, SignalType *sigType* = SignalType.SIGNAL, bool *isGlobal* = False)

creates a bus net object, using *sigType* to specify signal type for this bus net; *isGlobal* Boolean flag parameter is set to True, for a global net in the design.

find(string *name* = “”)

returns the bus net object having this name in the current DloGen design

getBaseName()

returns the base part of the name of this bus net (without bit indexing)

getBit(unsigned int *index*)

returns a bus net bit by its index

Feedback

getIndexStep()

returns the step of index of the bus net bits

getNumBits()

returns the number of bits of this bus net

getStartIndex()

returns the starting index of the bus net bits

getStopIndex()

returns the stopping index of the bus net bits

setBaseName(string *baseName*)

sets the base part of this bus net name

setIndexRange(unsigned int *startIndex*, unsigned int *stopIndex*)

sets the starting and stopping indexes of the bus net bits

BundleNet

A bundle net represents a logical connection by a multi-bit net composed of one or more scalar nets, bus nets, or a combination of both, each member net in a bundle net can repeat. The BundleNet class is derived from the Net class.

BundleNet(string *bundleName*, SignalType *sigType* = SignalType.SIGNAL, bool *isGlobal* = False)

creates a bundle net object, using the *bundleName* parameter. If this *bundleName* string parameter is empty or is the name of an existing net in the design, then an exception is raised.

find(string *name* = “”)

returns the bundle net object having this name in the current DloGen design

getMember(unsigned int *index*)

returns the specified member of this net at the specified bundle member index

getMembers()

returns a uniform list of the member nets which are contained in this bundle net

getName()

returns the name of this bundle net

Feedback

getNumMembers()

returns the number of members in this bundle net

getRepeat(unsigned int *index*)

returns the repeat count for the specified member of this bundle net

---

Pin

Represents the physical connection of terminals on a design instance to nets in the design. Note that a terminal can have more than one pin, so that multiple Pin objects can represent multiple physical connections, which are viewed as a single logical connection.

Pin(string *pinName*, string *termName*, Shape *shape*=None)

creates a Pin object for the specified terminal in the current design; *shape* parameter can be used to optionally associate a shape with this pin.

addInstPin(InstPin *instPin*, ShapeFilter *filter*=ShapeFilter())

uses shape references from *instPin* to create shapes associated with this pin

addShape(Shape *shapes*)

addShape(ulist[Shapes] *shapes*)

adds shape(s) to this pin; a shape can only be associated with a single pin

destroy()

destroys this pin, as well as the underlying OpenAccess database object

find(*name*='')

returns the pin having this name in the current DloGen design

getAccessDir()

returns access direction(s) for this pin (viewed from Instance containing this pin)

getAuthorPrefLayer()

returns layer which is the author-specified preferred layer for routing purposes

getBBox(ShapeFilter *filter*=ShapeFilter())

returns merged bounding box of geometries on layers specified by the *filter* ShapeFilter for which this pin has associated geometries

getLayers()

returns list of layers on which this pin has associated geometries

Feedback

getName()

returns the name for this pin

getNet()

returns the net which is associated with this pin

getPrefLayer()

returns author-specified preferred layer for this pin

getPrefLayerPropName()

returns name of property used to specify preferred routing layer for this pin

getShapes()

returns list of the shapes which are currently associated with this pin

getTerm()

returns the terminal associated with this pin

getTopLayer()

returns the top Layer on which this pin has associated geometries

removeShape(Shape *shapes*)

removeShape(ulist[Shapes] *shapes*)

removes the shape(s) from this pin

setAccessDir(Direction | ulist[Direction] *dir*)

sets access direction(s) for this pin (viewed from Instance containing this pin)

setAuthorPrefLayer(Layer *layer*)

sets layer to be author-specified preferred layer for routing purposes

setName(string *name*)

sets the name for this pin

setTerm(Term *term*)

associates terminal with pin; multiple pins may be associated with single terminal.

Attribute:

- **name** – the name of this pin (settable attribute)

Feedback

Term

Represents the logical connection points within a design. A terminal is logically associated with a net to export connectivity to the next higher-level in the design,

and is associated with one or more pins, which represent physical connection points.

Term(string *termName*, TermType *termType*=TermType.INPUT_OUTPUT)

creates new terminal in the current design; if there is already a net in the current design having the same name as this terminal, then the net will be connected to this terminal. Otherwise, a new net will be created, and connected to this terminal.

destroy()

destroys this terminal, first destroying any pins owned by this terminal

find(*name*='')

returns the terminal having this name in the current DloGen design

findCreate(string *name*)

returns the terminal having this name in the current DloGen design; if there is not any terminal in the current design having this name, then a new terminal is created.

getBit(unsigned int *index*)

for a single-bit terminal, always returns this Term object.

getMustJoinTerms()

returns list of all terminals which are in the “must-join” set of this terminal

getName()

returns the name for this terminal

getNet()

returns the net associated with this terminal object

getNumBits()

returns the number of bits of this Term object.

getPins()

returns list of all pins associated with this terminal

getTermType()

returns the terminal type value assigned for this terminal

Feedback

setMustJoin(Term *term*=None)

marks this terminal as requiring connection at an upper level of design hierarchy; if *term* parameter is None, then terminal will not be marked for connection.

setName(string *name*)

sets the name for this terminal; the name of any associated net for this terminal will also be changed, ensuring that nets and terminals will have the same name.

setNetOverride (string *assignmentName*, string *netName*)

creates an oaTermConnectDef associating this single-bit terminal with the oaAssignment named *assignmentName*. If the oaAssignment referred to does not exist, this single-bit terminal is associated with the net named *netName*. An exception is thrown if the net named *netName* does not exist.

setTermType(TermType *termType*)

sets the terminal type value for this terminal

Attributes:

- **name** – the name of this terminal (settable attribute)
- **type** – the terminal type for this terminal (settable attribute)
- **AttrType** – display attribute type for this terminal

BusTerm

A bus term represents a logical connection point within a design by one or more bits which are associated with common base name and vector range specification. The BusTerm class is derived from the Term class. Bus terms have vector names, for example, G[1:5:2].

BusTerm(string *baseName*, unsigned int *startIndex*, unsigned int *stopIndex*, unsigned int *indexStep*, TermType *termType* = TermType.INPUT_OUTPUT)

creates a bus term object, using the *baseName* parameter to name the base part (that is, without bit indexing) of this newly created BusTerm object

find(string *name* = “”)

returns the bus term object having this name in the current DloGen design

getBaseName()

returns the base part of the name of this bus term (without bit indexing)

Feedback

getBit(unsigned int *index*)

returns a bus term bit by its index

getIndexStep()

returns the step of index of the bus term bits

getNumBits()

returns the number of bits of this bus term

getStartIndex()

returns the starting index of the bus term bits

getStopIndex()

returns the starting index of the bus term bits

setBaseName(string *baseName*)

sets the base part of this bus term name.

setIndexRange(unsigned int *startIndex*, unsigned int *stopIndex*)

sets the starting and stopping indexes of the bus term bits

BundleTerm

A bundle term represents a logical connection by a multi-bit terminal composed of one or more scalar terminals, bus terminals, or a combination of both, each member terminal in a bundle terminal can repeat. The BundleTerm class is derived from the Term class. Bundle terminal have multi names, for example, 2*A,2*B[5],2*C[1:0].

BundleTerm(string *bundleName*, TermType *termType* = TermType.INPUT_OUTPUT)

creates a bundle terminal object, using the *bundleName* parameter. If the *bundleName* string parameter is empty, or is the name of an existing terminal in the design, then an exception is raised.

find(string *name* = “”)

returns the bundle term object having this name in the current DloGen design

getMember(unsigned int *index*)

returns the specified member of this terminal at the specified bundle member index

getMembers()

returns a uniform list of the member terminals which are contained in this bundle terminal

Feedback

getName()

returns the name of this bundle terminal

getNumMembers()

returns the number of members in this bundle terminal

InstTerm

The InstTerm class is used to represent the connection between a net and a terminal of the master of an instance in the design. These instance terminals will automatically be created, whenever an instance is created within a design.

find(string *name*, Instance *inst*)

returns the instance terminal having this name in the current DloGen design

findInstPin(string *name*="")

returns instance pin having this name for this instance terminal

getBit(unsigned int *index*)

returns the InstTerm object that corresponds to the specified index bit of this instance terminal.

getInst()

returns the Instance object associated with this instance terminal

getInstPins()

returns list of all of the InstPin objects for this instance terminal

getMustJoinInstTerms()

returns list of all InstTerm objects which must be joined together

getNet()

returns the net associated with this instance terminal

getNumBits()

returns the number of bits of this InstTerm object.

getTermName()

returns the name for this instance terminal

getTermType()

returns the terminal type value for this InstTerm object

Feedback

setNet(Net *net*)

adds this instance terminal to the specified net; if the *net* parameter is None, then this instance terminal is removed from any net to which it is connected.

Attributes:

- **termName** – the name of this instance terminal
- **AttrType** – display attribute type for this instance terminal

---

InstPin

The InstPin class is used to represent a mapping of the pin in the instance master into the current coordinate system. Since this InstPin object is simply a mapping, these InstPin objects are read-only, and created whenever an instance is created in the current design. Note that there is not any OpenAccess equivalent object for this InstPin class object.

find(string *name*, Instance *inst*)

find(string *name*, InstTerm *instTerm*)

returns the instance pin having this name in the current DloGen design; these instance pins are obtained from the *inst* Instance or the *instTerm* InstTerm object.

getAuthorPrefLayer()

returns author-specified preferred layer for routing purposes for the corresponding pin in the instance submaster.

getBBox(ShapeFilter *filter*=ShapeFilter())

returns the bounding box fpor the corresponding pin in the instance submaster, for all layers on which this pin has associated geometries.

getInst()

returns the instance for this instance pin

getInstTerm()

returns the instance terminal for this instance pin

getLayers(LayerList *layers*=None)

returns list of layers on which the corresponding pin in the instance submaster has associated geometries.

getNet()

returns the net for this instance pin

Feedback

getPinAccessDir()

returns the access direction(s) for the corresponding pin in the instance submaster

getPinName()

returns the pin name for this instance pin

getPrefLayer()

returns author-specified preferred layer for routing purposes for the corresponding pin in the instance submaster; if there is no preferred layer, the top layer is used.

getShapeRefs(ShapeFilter *filter*=ShapeFilter())

returns list of shape references for all shapes which are associated with the corresponding pin in the instance submaster.

getTermName()

returns the name of the terminal for this instance pin

getTermType()

returns the terminal type value for this InstPin object

getTopLayer()

returns the top layer on which the corresponding pin in the instance submaster has associated geometries, as defined by the layer number in the Technology file.

Attribute:

- **pinName** – the pin name for this instance pin

RouteTarget

Class used by the RoutePath class to specify shapes which should be connected by a route path; represents a target to "route from" or "route to" by RoutePath methods. RouteTarget object consists of one or more route boxes, along with associated layers. Note that any Rect, RectRef, Pin or InstPin objects will automatically be converted to appropriate RouteTarget object.

RouteTarget(Rect *rect*, Point *point*=None)

RouteTarget(RectRef *rectRef*, Point *point*=None)

RouteTarget(InstPin *instPin*, Point *point*=None)

Feedback

RouteTarget(Pin *pin*, Point *point*=None)

getBBox(ShapeFilter *filter*=ShapeFilter())

returns bounding box for the route boxes contained in this RouteTarget

getChosenAccessDir()

returns access direction which was chosen for this RouteTarget

getChosenAccessPoint()

returns access point which was chosen (or specified) for this RouteTarget

getChosenBox()

returns box for chosen shape for this RouteTarget

getChosenLayer()

returns layer on which the chosen route box is defined for this RouteTarget

getLayers()

returns list of layers on which any route boxes are defined

getName()

returns name generated for this RouteTarget

getPrefLayer()

returns name of any preferred layer

getRoutePathIntersectBox()

returns box which is intersection of route box and RoutePath

getRoutePathLayer()

returns layer which will be used by the associated RoutePath

hasLayer(Layer *layer*)

returns True if RouteTarget has a route box on specified layer

isSingleBox()

returns True if RouteTarget has a single route box

isValid()

returns True if RouteTarget is valid

needsContact()

returns True if RouteTarget requires contact to be generated

Feedback

Topology

Provides class used to store a string used to describe basic topological configuration for a parameterized cell design object.

Topology(string *topology*)

creates Topology object using specified topology string

getName()

returns string value for this Topology object

setName(string *name*)

sets string value for this Topology object

Connectivity Reference Classes

Hierarchical Object Reference classes are used to reference connectivity objects in the lower-level hierarchy in a hierarchical layout design. The Python API provides three major classes for the handling of connectivity information of the lower-level hierarchy of a layout design: NetRef, PinRef, and TermRef.

BlockObjectRef

Base class used for all connectivity Hierarchical Object Reference classes which allows any connectivity object contained within the current design to be used as a reference object.

getProps()

returns a Python dictionary-like object (the PropSet object), which contains any

properties which have been defined for this BlockObjectRef object

Attribute:

props – properties and values defined for this BlockObjectRef object

NetRef

Derived from the base BlockObjectRef class and used whenever the referenced object is a net.

getBit(unsigned int *index*)

for a single-bit net, always returns this NetRef object; for a bus net, returns a busnet bit by its index

getName()

returns the name for this NetRef object

getNumBits()

returns the number of bits of this NetRef

getPinRefs()

returns a list of PinRefs which are connected to any terminal associated with thisNetRef object

getShapeRefs(ShapeFilter *filter*=ShapeFilter()) – returns a uniform list of the

ShapeRefs which are currently associated with this NetRef

getSignalType()

returns the signal type value assigned to this NetRef object

getTermRef()

returns any terminal that is associated with this NetRef

getViaRefs()

returns a list of all ViaRefs which are associated with this NetRef

isGlobal()

returns the Boolean flag value indicating that this NetRef is a global net

Attribute:

- **globalNet** – value of the Boolean flag indicating that this is a global net

TermRef

Derived from the base BlockObjectRef class and used whenever the referenced object is a term.

getName()

returns the name for this TermRef object

getNetRef()

returns the NetRef associated with this TermRef terminal object

getPinRefs()

returns a uniform list of all of the PinRef objects associated with this TermRef terminal object

Feedback

getTermType()

returns the terminal type value assigned for this TermRef object

PinRef

Derived from the base BlockObjectRef class and used whenever the referenced object is a pin.

getAccessDir()

returns the access direction for this PinRef

getName()

returns the name for this PinRef object

getNetRef()

returns the NetRef object which is associated with this PinRef

getShapeRefs(ShapeFilter *filter*=ShapeFilter())

returns a uniform list of the ShapeRefs which are currently associated with this PinRef

getTermRef ()

returns the terminal associated with this PinRef object

Utility Classes

Basic classes provided to facilitate various parameterized cell design creation tasks. These tasks include specification of parameters, handling of parameter values, user-defined properties for different design objects, handling lists of points, generation of unique names and log file management. These classes include the ParamArray, ParamSpecArray, PropSet, PointList, Log and Unique classes.

ParamArray

Stores different parameters and their values for a parameterized cell design. Class objects can be used much like built-in Python dictionary.

ParamArray()

ParamArray(**argv*)

creates ParamArray object; if lists are used, must be of form “name, type, value”

Feedback

ParamArray(**_kwds_)

creates ParamArray object; if keywords are used, must be of form “name=value”

add(_name_, _value_)

add(_name_ = _value_)

adds a parameter and its value to this ParamArray

get(string _name_)

returns value for parameter with specified name; if no parameter, exception raised

has_key(string _name_)

returns True if there is a parameter of this name in this ParamArray

iteritems()

returns iterator for parameter and value pairs (items) in this ParamArray

iterkeys()

returns iterator for parameters (keys) in this ParamArray

itervalues()

returns iterator for values (values) in this ParamArray

remove(string _name_)

removes parameter and its value from this ParamArray

reset()

resets this ParamArray, by removing all parameters and values

set(_name_, _value_)

set(_name_ = _value_)

sets value for parameter to given value; parameter will be added, if does not exist

setFromSpecs(ParamSpecArray _specs_)

copies all parameters from the _specs_ ParamSpecArray

update(ParamArray _params_)

updates all parameters using values from _params_ ParamArray; if parameter already exists, its value will be updated, otherwise parameter and its value will be added.

Feedback

ParamSpecArray

Stores different parameter specifications for a parameterized cell design. Class objects can be used much like built-in Python dictionary.

add(name, value, docString=None, constraint=None)

adds parameter and its specification to this ParamSpecArray

has_key(string name)

returns True if there is a parameter of this name in this ParamSpecArray

iteritems()

returns iterator for parameters and specifications (items) in this ParamSpecArray

iterkeys()

returns iterator for all of the parameters (keys) in this ParamSpecArray

itervalues()

returns iterator for all of the specifications (values) in this ParamSpecArray

remove(string name)

removes named parameter and its specification from this ParamSpecArray

verify(ParamArray params)

verifies parameters in ParamArray, using specifications from this ParamSpecArray

Attribute:

- **tech** – returns technology object used when ParamSpecArray was created

ViaParam

Stores parameter values to create a standard via, using pre-defined parameter names. Class objects can be used much like built-in Python dictionary.

ViaParam(ViaParam *params* = None, ulist[Coord] *layer1Ext*=None, ulist[Coord] *layer2Ext*=None, ulist[Coord] *implant1Ext*=None, ulist[Coord] *implant2Ext*=None, ulist[Coord] *layer1Offset*=None, ulist[Coord] *layer2Offset*=None, ulist[Coord] *originOffset*=None, ulist[Coord] *cutSpace*=None, ulist[Coord] *cutSize*=None, Layer *cutLayer*=None, ulist[int] *NumHVCuts*=None)

creates ViaParam object, using the specified parameter values; if *params* is specified, then ViaParam object obtains its values from *params* object.

Feedback

hasDefault(string *paramName*)

returns True if *paramName* is set to its default value, and returns False otherwise

items( )

returns list of all parameters and values (items) defined for this ViaParam

iteritems( )

returns iterator for parameters and values (items) in this ViaParam

keys( )

returns list of all parameters (keys) defined for this ViaParam

iterkeys()

returns iterator for all of the parameters (keys) in this ViaParam

setDefault(string *paramName*)

sets value for *paramName* parameter to its default value

Attributes:

- **cutLayer** – cut layer for the standard via
- **cutSize** – via cut sizes used by the standard via
- **cutSpace** – spacing values between via cuts for the standard via
- **implant1Ext** – enclosure values for implant layer over bottom layer (layer1)
- **implant2Ext** – enclosure values for implant layer over top layer (layer2)
- **layer1Ext** – enclosure values for bottom layer (layer1) over the cut layer
- **layer1Offset** – x and y offset values for bottom layer (layer1) enclosure rectangle
- **layer2Ext** – enclosure values for top layer (layer2) over the cut layer
- **layer2Offset** – x and y offset values for the top layer (layer2) enclosure rectangle
- **numHVCuts** – number of horizontal and vertical cuts for the standard via
- **originOffset** – offset in x and y coordinates from the origin for the standard via

---

PropSet

Provides storage for user-defined properties which have been defined for a given design object. Such properties can be defined and associated with the DloGen,

Feedback

PhysicalComponent and Lib class objects (or any objects derived from these classes, except for the Grouping class and objects derived from the Grouping class).

clear()

clears this PropSet, by removing all property definitions

empty()

returns True if this PropSet is empty

get(string *name*, object *defaultVal*=None)

returns value for property with given name; if no property exists, default value used

getProp(string *name*)

returns value for property with given name; if no property exists, exception raised

has_key(string *name*)

items()

returns list of property name and value pairs (items) in this PropSet

iteritems()

returns iterator for property name and value pairs (items) in this PropSet

iterkeys()

returns iterator for property names (keys) in this PropSet

itervalues()

returns iterator for property values (values) in this PropSet

keys()

returns list of property names (keys) in this PropSet

update(PropSet *other*)

updates this PropSet, by adding all of the property names (keys) and property values (values) contained in *other* PropSet; this operation may overwrite existing property names and property values in this PropSet.

values()

returns list of property values (values) in this PropSet

Feedback

PointList

Class used to define and operate on lists of points. These PointList class objects are typically used with the various Shape class creation methods, such as the Polygon or Path class creation methods, as well as the MultiPath creation method.

PointList(ulist[Point]=None)

creates PointList object using the specified list of points

addOffsets(Coord *begin*=0, Coord *end*=0)

adds *begin* offset to first point and *end* offset to last point in this PointList object

compress(bool *isClosed*=True)

compresses this PointList object, by removing any coincident or collinear points

containsPoint(Point *point*, bool *incEdges*=True)

returns True if *point* is contained inside point list shape, and False otherwise

copy()

returns copy of this list of points

deepcopy()

returns deep copy of this list of points

genJustifyPoints(Direction *justify*, Coord *sep*)

returns justified point list constructed from this PointList object

getArea()

returns area enclosed by point list shape. Note this area value will be positive for counter-clockwise point ordering, negative for clockwise ordering, zero otherwise.

getBBox()

returns bounding box for this list of points

hasExtraPoints(bool *isClosed*=True)

returns True if the points in this list contains any coincident or collinear points

isOrthogonal(bool *isClosed*=True)

returns True if the points in this list are orthogonal, otherwise False

isSelfIntersecting(bool *isClosed*=True)

returns True if point list shape is self-intersecting, and False otherwise.

Feedback

mirrorX(Coord *xCoord*=0)

mirrors all points around the X-coordinate axis

mirrorY(Coord *yCoord*=0)

mirrors all points around the Y-coordinate axis

moveBy(Coord *dx*, Coord *dy*)

moves all points by the specified x and y coordinate values

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER)

moves all points so that the specified *destination* point becomes the *handle* point for the bounding box which contains these points.

moveTowards(Direction *dir*, Coord *distance*)

moves all points in the specified direction by the specified distance

onEdge(Point *point*, bool *isClosed*=True)

returns True if *point* is on edge of point list shape, and False otherwise

rotate90(Point *origin*=None)

rotates all points 90 degrees, in a counter-clockwise direction

rotate180(Point *origin*=None)

rotates all points 180 degrees, in a counter-clockwise direction

rotate270(Point *origin*=None)

rotates all points 270 degrees, in a counter-clockwise direction

transform(Transform *trans*)

applies the specified transform to all points

Log

Class used to write information to the various logging files maintained by the Santana design system, such as errors, warnings, general information, debug and test data.

debug(string *msg*)

writes specified message string to the debug log file

error(string *msg*)

writes specified message string to the error log file

Feedback

info(string *msg*)

writes specified message string to the info log file

test(string *msg*)

writes specified message string to the test log file

warning(string *msg*)

writes specified message string to the warning log file

Unique

Generates unique name for name within the name space of a given design object. The generated name is guaranteed to be unique within a single design, provided that this class is used to generate all names for design objects within the current design.

Name(string baseName)

returns unique name, based upon the specified base name (defaults to “CNI_”)

NameMapper

Class used to map names for instances or nets when the **clone**() method is used. This name mapping can be defined using a suffix-prefix string having the general form “subPrefix/addPrefix:subSuffix/addSuffix”, a Python dictionary, or a general Python callback function. This name mapping object can then be passed as an optional parameter to the **clone**() method

map(string *name*)

method used to perform name mapping, returns the mapped name

SnapType

Defines snap types which can be used with Grid object or physical components.

SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC and SnapType.ROUND_CEIL used for SnapType.**ceil**(), SnapType.**floor**(), SnapType.**round**(), SnapType.**trunc**() and SnapType.**round_ceil**().

snap(*self*, Coord *size*, Coord *val*)

performs snap operation using *self* snapping type

ceil(Coord *size*, Coord *val*)

performs snap operation, using CEIL ceiling rounding convention

Feedback

floor(Coord *size*, Coord *val*)

performs snap operation, using FLOOR floor rounding convention

round(Coord *size*, Coord *val*)

performs snap operation, using ROUND round rounding convention

round_ceil(Coord *size*, Coord *val*)

performs snap operation, using ROUND_CEIL rounding convention

trunc(Coord *size*, Coord *val*)

performs snap operation, using TRUNC truncation rounding convention

---

Grid

Abstract representation of the manufacturing grid used to manufacture integrated circuits. User can specify Grid sizes for both X and Y coordinate values.

Grid(Coord *xSize*, Coord *ySize*=None, SnapType *snapType*=SnapType.CEIL)

creates Grid object, based upon specified sizes and snap type

getSize()

returns size or resolution value

getSnapType()

returns snap type defined for this Grid object

getXSize()

returns X-coordinate size or resolution value

getYSize()

returns Y-coordinate size or resolution value

setSize(Coord *size*)

sets size or resolution value for this Grid object

setSnapType(SnapType *snapType*)

sets the snap type for this Grid object

setXSize(Coord *size*)

sets X-coordinate size or resolution value for this Grid object

Feedback

setYSize(Coord *size*)

sets Y-coordinate size or resolution value for this Grid object

snap(Coord *val*, SnapType *snapType*=None, unsigned int *mult*=1)

returns “snapped to” grid coordinate value, based upon snap type

snapX(Coord *val*, SnapType *snapType*=None, unsigned int *mult*=1)

returns “snapped to” X-coordinate value, based upon snap type

snapY(Coord *val*, SnapType *snapType*=None, unsigned int *mult*=1)

returns “snapped to” Y-coordinate value, based upon snap type

---

Numeric

The Numeric class is used to create a floating point number from a string representation (such as “10ns”), using a floating point number along with a pre-defined scaling factor. These pre-defined scaling factors are one of the following:

Character	Name	Multiplier
Y	Yotta	1e24
Z	Zetta	1e21
E	Exa	1e18
P	Peta	1e15
T	Tera	1e12
G	Giga	1e09
M	Mega	1e06
K or k	Kilo	1e03
‘’	No Scale Factor	1.0
%	percent	1e-2
c	centi	1e-2
m	milli	1e-3
u	micro	1e-6
n	nano	1e-9

Feedback

p	pico	1e-12
f	femto	1e-15
a	atto	1e-18
z	zepto	1e-21
y	yocto	1e-24

Since this Numeric class is derived from the base Python float class, it can be used just like a regular floating point number in any numerical computation.

Numeric(float | int | string)

creates Numeric object, using specified number or string value; the string value must be a number followed by one of the pre-defined scaling factors.

scaleFormat(string *scaleFactor*=None)

returns floating point number formatted using specified *scaleFactor* scaling value

Attributes:

- **scaleFactor** – original scale factor used when Numeric object created
- **scale_factors** – list of all available scaling factors, along with their values

cFloat

The cFloat class is used to create a single precision floating point number, which has the same precision as a C/C++ “float” type floating-point number. This class can be used with parameterized cells which were originally developed in SKILL and use parameters with single precision floating point numbers.

cFloat(float)

creates a cFloat object, based upon the specified floating point number

AttrDict

The AttrDict class is used to create a Python dictionary in which the elements of the dictionary can directly be accessed through the use of attribute syntax. For example, instead of using “dict[key]” to access dictionary values, “dict.key” can be used instead.

Feedback

AttrDict([arg])

Creates an AttrDict dictionary object, based upon the specified keys and values. Keys and values can be specified using positional arguments or a set of keyword arguments, such as “AttrDict(a=1, b=2)” or “AttrDict([[‘a’,1], [‘b’,2]])”.

clear()

removes all items from this attribute dictionary

get(*k*, *d*=None)

returns value for specified *key*; if key is not found, returns specified *default* value

has_key(*k*)

returns True if *k* is a key in this attribute dictionary, and False otherwise

items()

returns all of the key/value pairs defined for this attribute dictionary

iteritems()

returns iterator for all of the key/value pairs in this attribute dictionary

iterkeys()

returns iterator for all of the keys in this attribute dictionary

itervalues()

returns iterator for all of the values in this attribute dictionary

keys()

returns all of the keys defined for this attribute dictionary

pop(*k*, [*default*])

removes specified key *k* from dictionary, and its corresponding value is returned; if key *k* does not exist, then an exception will be raised.

popitem()

removes and returns an arbitrary key/value pair from this attribute dictionary; can be used to destructively iterate over dictionary, and remove its items one by one.

setdefault(*k*, [*default*])

if specified key *k* exists, then return its value; if key *k* is not in dictionary, then insert *key* into dictionary. If *default* is specified, then it is used as the value for key *k*; otherwise, the value for key *k* is set to None.

Feedback

update(*E*)

updates this dictionary with all key/value pairs from the specified dictionary *E*

values()

returns all of the values defined for this attribute dictionary

---

OrderedDict

The OrderedDict class is used to create a Python dictionary in which the elements of the dictionary are kept ordered according to the order in which the keys were specified.

OrderedDict(*init_val*=(), *strict*=False) – creates an OrderedDict dictionary object,

based upon the specified *init_val* initial value ordered list of key and value pairs.

clear()

removes all items from this ordered dictionary

copy()

returns a copy of this ordered dictionary

index(*key*)

returns position of the specified *key* in this ordered dictionary.

insert(*index, key, value*)

sets specified *value* for specified *key* at specified position *index* in this dictionary

items()

returns list of tuplesrepresenting all key/value pairs for this ordered dictionary

iteritems()

returns iterator for all of the key/value pairs in this ordered dictionary

iterkeys()

returns iterator for all of the keys in this ordered dictionary

itervalues()

returns iterator for all of the values in this ordered dictionary

keys()

returns all of the keys defined for this attribute dictionary

Feedback

pop(*key*, **args*)

removes specified *key* from dictionary, and its corresponding value is returned; if *key* does not exist, then an exception will be raised.

popitem(*i* = -1)

removes and returns specified key/value pair from this attribute dictionary; can be used to destructively iterate over dictionary, and remove its items one by one.

rename(*old_key*, *new_key*)

renames key, without modifying sequence order for this ordered dictionary

reverse()

reverses sequence order for all items in this ordered dictionary

setdefault(*key*, *defval*=None)

if specified *key* exists, then return its value; if *key* is not in dictionary, then insert *key* into dictionary. If *defval* is specified, then it is used as the value for *key*; otherwise, the value for *key* is set to None.

setitems(*items*)

uses specified *items* to set all of the items for this ordered dictionary

setkeys(*keys*)

uses specified *keys* to replace all of the keys for this ordered dictionary

setvalues(*values*)

uses specified *values* to replace all of the values for this ordered dictionary

sort(**args*, ***kwargs*)

sorts key order for this ordered dictionary, same as Python list **sort**() method

update(*from_od*)

updates ordered dictionary with key/value pairs from specified ordered dictionary

values()

returns list of all values defined for this ordered dictionary

---

ParamDictSpec

The ParamDictSpec class is used to specify parameters and specifications for these parameters. This is done by creating an ordered Python dictionary in which the keys are the parameter names, and the values are the specifications for these parameters.

Feedback

ParamDictSpec()

ParamDictSpec derives from OrderedDict base class, so uses same creation method

match(data)

checks that parameter and value matches parameter specification

optional(*key, datatype, doc*=None, *constraint*=None)

adds optional parameter and specification to this ParamDictSpec object

required(*key, datatype, doc*=None, *constraint*=None)

adds required parameter and specification to this ParamDictSpec object

---

CDF

The CDF class is used to read CDF data which is stored as a property on a cell. This CDF data is returned using an AttrDict attribute dictionary; all CDF data can then be accessed using standard attribute syntax. Note that there is no need to use a creation method, since this CDF data is already being stored as a property on the cell.

forCell(string *libName*, string *cellName*)

returns stored CDF data for this cell, using an attribute dictionary; this allows all CDF data to be accessed using standard attribute syntax.

DPL

The DPL class is used to convert a SKILL Disembodied Property List into an AttrDict attribute dictionary. This allows SKILL data which is stored using disembodied property lists to be more easily accessed using a Python dictionary. Note that this class only contains a creation method, since once the attribute dictionary is created, the DPL data can be directly accessed.

DPL([arg])

returns data for the SKILL disembodied property list, using an attribute dictionary; this allows all DPL data to be accessed using standard attribute syntax.

Feedback

Fill

The Fill class is used to fill a physical component with patterns of either rectangles or instance array objects. The filling methods in this Fill class are “smart” methods which make use of the Santana “Geometry Engine” to perform the polygon filling operations,

so that they will make use of the relevant design rules in the associated technology file.

fillPhysCompWithInstArrays(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, InstanceArray *instArray*, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None)

Fills *comp* physical component with *instArray* instance array.

fillPhysCompWithLargeArraysOfInstArrays(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, Coord *arraySpaceX*, Coord *arraySpaceY*, InstanceArray *instArray*, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None)

Fills *comp* physical component with a large array of instance arrays; the *arraySpaceX* and *arraySpaceY* values specify spacing between instance arrays.

fillPhysCompWithLargeArrayOfRects(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, Coord *arraySpaceX*, Coord *arraySpaceY*, unsigned *numRows*, unsigned *numCols*, Coord *width*=None, Coord *height*=None, Coord *spaceX*=None, Coord *spaceY*=None, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None)

Fills *comp* physical component with large array of rectangles. The *numRows* and *numCols* values specify dimensions of this array of rectangles, while *arraySpaceX* and *arraySpaceY* values specify spacing between rectangles in the array.

fillPhysCompWithRects(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, Coord *width*=None, Coord *height*=None, Coord *spaceX*=None, Coord *spaceY*=None, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None, GapStyle *gapStyle* = GapStyle.MIN_CENTER)

Fills *comp* physical component with equally sized and spaced rectangles

CrossOver

The CrossOver class computes the minimum 45 degree X cross-over of two parallel paths, which may have different widths. This **CrossOver** class provides methods to generate the polygons which cross over and connect each of these two parallel paths. These 45 degree X cross-overs will be computed based upon the widths of the two parallel paths, as well as the spacing between them.

Feedback

CrossOver(Coord *width1*, Coord *space*, Coord *width2*=None, Grid *grid*=None, Coord *space45*=None, Coord *width45*=None)

creates minimum 45 degree X cross-over between two parallel paths, based upon specified widths of the two paths, as well as the spacing between them.

genPath1Polygon(Direction *dir*, Direction *justify*, Point *innerPoint*, Layer *layer*)

generates polygon for cross-over points returned by **getPath1Points**() method

genPath2Polygon(Direction *dir*, Direction *justify*, Point *innerPoint*, Layer *layer*)

generates polygon for cross-over points returned by **getPath2Points**() method

getDistance()

returns minimum distance needed to create 45 degree X cross-over obtained by connecting parallel paths on one side of the cross-over to the other side.

getPath1Extends()

returns list of two cross-over extensions for first path

getPath1Points(Direction *dir*, Direction *justify*, Point *innerPoint*)

returns list of two point lists for cross-over points for first path; first point list is list of points for inner edge points, while second is list of points for outer edge.

getPath2Extends()

returns list of two cross-over extensions for second path

getPath2Points(Direction *dir*, Direction *justify*, Point *innerPoint*)

returns list of two point lists for cross-over points for second path; first point list is list of points for the inner edge points, while second is list of points for outer edge.

getSpace()

returns spacing value between first path and second path

getWidth1()

returns width of first path

getWidth2()

returns width of second path

getWidth45()

returns actual diagonal width value, which is scaled by the square root of two

Feedback

set(Coord *width1*, Coord *space*, Coord *width2*=None, Grid *grid*=None, Coord *space45*=None, Coord *width45*=None)

changes dimensions of **CrossOver** object, using specified parameter values; parameter values are exactly the same as parameter values in creation method.

Attributes:

- **distance** – minimum distance required to create cross-over
- **path1Extends** – list of cross-over extensions for first path
- **path2Extends** – list of cross-over extensions for second path
- **width45** – actual diagonal width value

Marker

The Marker class indicates design violations and the objects causing these design violations in the OpenAccess database. The location of the design violation is represented using a list of points assigned to the marker. In addition, a message string which describes the design violation can be attached to the marker, along with the objects which caused the design violation. The name of the tool which reported the design violation, along with the severity level for the design violation can also be assigned to the marker.

Marker(ulist[Point] *points*, string *msg*="Error marker", string *shortMsg*="Error marker", string *tool*="*PyCell*", bool *isVisible*=True, bool *isClosed*=True, MarkerSeverity *severity*=MarkerSeverity.FATAL_ERROR)

creates a Marker object, based upon specified points for the location of the design violation, strings for error messages and tool, as well as marker severity level.

addObject(PhysicalComponent | BlockObject | ulist[PhysicalComponent] | ulist[BlockObject] *obj*)

adds specified physical component(s) or BlockObject(s) to this marker

clone()

returns a cloned copy of this Marker object

getBBox(ShapeFilter *filter*=ShapeFilter())

returns bounding box for this marker

getMsg()

returns message string describing design violation for this marker

Feedback

getObjects()

returns list of objects causing design violations associated with this marker

getPoints()

returns list of points for area or location of the design error for this marker

getSeverity()

returns severity level of design violation for marker, using pre-defined values: ANNOTATION, INFO, ACKNOWLEDGED_WARNING, WARNING, SIGNED_OFF_ERROR, ERROR, SIGNED_OFF_CRITICAL_ERROR, CRITICAL_ERROR or FATAL_ERROR.

getShortMsg()

returns short message string associated with this marker

getTool()

returns name of tool reporting design violation for this marker

isClosed()

returns True if list of points for this marker is closed; returns False, otherwise

isVisible()

returns True if this marker should be visible; returns False, otherwise

removeObject(PhysicalComponent | BlockObject | ulist[PhysicalComponent] | ulist[BlockObject] *obj*)

removes specified physical component(s) or BlockObject(s) from this marker

setIsClosed(bool *val*)

indicates whether the list of points for this marker is closed or not

setIsVisible(bool *val*)

indicates whether this marker should be visible or not.

setMsg(string *msg*)

sets message string describing design violation for this marker

setPoints(ulist[Point])

sets list of points which specifies area or location of design error for this marker

setSeverity(MarkerSeverity)

sets severity level of design violation for this marker, using pre-defined values: ANNOTATION, INFO, ACKNOWLEDGED_WARNING, WARNING,

SIGNED_OFF_ERROR, ERROR, SIGNED_OFF_CRITICAL_ERROR, CRITICAL_ERROR or FATAL_ERROR.

setShortMsg(string *msg*)

sets short message string to be associated with this marker.

setTool(string *msg*)

sets name of tool reporting design violation for this marker

DrcSummary

The DrcSummary class is used to obtain information about the results of running the design rule checking program, using the DloGen "**fgDrc**()" method.

checkedRules()

returns number of design rules which were checked; this number is calculatedafter reduction of the design rules in the DRC rule deck has taken place.

errorNumber(string *ruleName*)

returns number of errors generated for specific named design rule

getRuleList()

returns list of names of all design rules which failed; these design rule name strings can then be used with the "errorNumber()" method.

totalErrorNumber()

returns total numbers of design rule errors found for the current design

---

AppDef

The AppDef class is used to store extension values to existing database objects that are used to create extensions on Shape, Instance, Via, Net, Term, Pin, and InstTerm objects. This AppDef class has no separate constructor for an object, and it is only necessary to call thr get(), set(), and destroy() methods to access the extension values on existing database objects.

destroy(*object*, string *name*)

destroys AppDef data with the *name* for the specified *object*.

get(*object*, string *name*)

returns the value of the AppDef with the *name* from the specified *object*. If there is no AppDef with this *name* on the *object* , then an exception is raised.

Feedback

set(*object*, string *name*, string *valueType*, *value*)

sets AppDef data according to the given *valueType* and *value* for the specified *object*. The *valueType* supports the following extension types:- boolean: Stores a boolean value on database objects.- int: Stores an integer value on database objects.- float: Stores a floating point number on database objects.- double: Stores a double precision floating point number on database objects.- string: Stores a string value on database objects.

3

Global Functions

fgDrc(Dlo *dlo*, DrcSettings *drcSettings*=None)

Runs the design rule checking program on the specified *dlo* design object. The returned value is the *drcSummary* utility class object which can be used to query the results of running design rule checking; methods are provided to determine the number of errors, the number of design rules checked, etc. The *drcSettings* default parameter is only provided for future enhancements; it is not necessary to use this parameter, as the default will automatically be used when calling this function. This function uses all of the design rules in the Santana technology file attached to the specified *dlo* design object, as is done when running the DRC program through the graphical environment.

stretchHandle(Shape *shape*, string *name*, string *parameter*, Location *location*, Direction *direction*, string *display* = “”, float *minVal* = None, float *maxVal* = None, string *stretchType* = “relative”, string *userSnap* = “0.005”, string *userScale* = “1.0”)

Assigns graphical stretch handle to shape in the generated PyCell layout. The *shape* parameter is the actual shape object; the *name* parameter is a unique name for the stretch handle, while *parameter* is the name of the PyCell parameter being modified by the user. Note that this *shape* parameter can also be an instance or the current design object; for an instance, the *name* parameter should be the name of the instance and for the current design object, the *name* parameter will be ignored. In the case of a hierarchical PCell, the *shape* parameter can also be a reference to a shape in the PCell hierarchy, that is a ShapeRef object. In this case, the stretch handle property will be created on the corresponding instance object which contains the referenced shape. The *location* parameter specifies which location on the *shape* bounding box should be used to attach the stretch handle, while the *direction* parameter specifies the stretching directions (NORTH_SOUTH or EAST_WEST). The *minVal* and *maxVal* parameters are the minimum and maximum values for the PyCell parameter. The *stretchType* parameter is either “relative” or “absolute” and specifies how the stretched distance should be measured. The *userSnap* parameter is the resolution value used for snapping parameter values, while the *userScale* is the scale factor used to multiply the change in PyCell parameter value. The optional *display* parameter can be used to display a specified text string on the Text layer at the location of the stretch handle location point.

stretchHandleCustom(Shape *shape*, string *name*, string *parameter*, Location *location*, Direction *direction*, string *display* = "", **kwargs)

Assigns graphical stretch handle to shape in the generated PyCell layout. The *shape* parameter is the actual shape object; the *name* parameter is a unique name for this stretch handle, while *parameter* is the name of the PyCell parameter being modified by the user. Note that this *shape* parameter can also be an instance or the current design object; for an instance, the *name* parameter should be the name of the instance and for the current design object, the *name* parameter will be ignored. In the case of a hierarchical PCell, the *shape* parameter can also be a reference to a shape in the PCell hierarchy, that is a ShapeRef object. In this case, the stretch handle property will be created on the corresponding instance object which contains the referenced shape. The *location* parameter specifies which location on the bounding box for the shape should be used to attach the stretch handle, while the *direction* parameter specifies the directions in which the shape may be stretched (NORTH_SOUTH or EAST_WEST). The optional *display* parameter can be used to display a specified text string on the Text layer at the location of the stretch handle location point. This **stretchHandleCustom**() function simply copies any optional keyword arguments and values which may be specified, and then appends them to the *pycStretch* string property which is associated with the shape in the PCell layout. It is then left for the EDA tool application to interpret these additional keyword arguments and values and perform any indicated operations.

autoAbutment(Shape *shape*, float *pinSize*, ulist[Direction] *directions*, string *abutClass*, list *abut2PinBigger*, list *abut3PinBigger*, list *abut2PinEqual*, list *abut3PinEqual*, list *abut2PinSmaller*, list *abut3PinSmaller*, list *noAbut*, string *function* = “”)

Defines auto-abutment properties for a PyCell and attaches them to shapes in the generated layout. The *shape* parameter is the pin shape object which can be abutted, while the *pinSize* specifies the width of this pin shape. The *directions* parameter specifies the list of directions in which this pin shape may be abutted, while the *abutClass* parameter is a string which identifies the abutment class. The next seven parameters specify values for different types of abutment conditions, using a fixed format. These abutment conditions are categorized by the number of pin shapes which are connected to the same net (either 2 or 3 devices connected to the same net), as well as the relative size of the pin shapes which are being abutted. The final optional *function* parameter specifies the name of any external abutment function which should be called. Note this function will set all required pin shape auto-abutment properties, except for *pycShapeName*; this property should instead be set using the Shape **setName**() method for the pin shape.