Piecad Cheatsheet

Info Functions

`version() -> 'str'`
Piecad version

Bulk Operations

`compose(*objs: Obj2d | Obj3d) -> Obj2d | Obj3d`
Returns a single object containing all the disjoint objects.

`difference(*objs: Obj2d | Obj3d) -> Obj2d | Obj3d`
Returns the object removing the second through the last objects from the first object.

`hull(*objs: Obj2d | Obj3d) -> Obj2d | Obj3d`
Return a convex hull of the given objects.

`hull_points(pts: list[tuple[float, float]] | list[tuple[float, float, float]]) -> Obj2d | Obj3d`
Return a convex hull of the given list of points.

`intersect(*objs: Obj2d | Obj3d) -> Obj2d | Obj3d`
Returns the object made by adding those portions that occur only in all `objs` together.

`union(*objs: Obj2d | Obj3d) -> Obj2d | Obj3d`
Returns the object made by adding all the `objs` together.

2d Primitives

`circle(radius: float, segments: int = -1) -> Obj2d`
Make a circle of a given radius.

`ellipse(radii: list[float, float], segments: int = -1) -> Obj2d`
Make an ellipse with the given radii.

`path(initial_point: tuple[float, float] = (0, 0), segments: int = -1) -> object`
Create a Path object containing an SVG-like path starting at `initial_point`.

`polygon(paths: list[list[tuple[float, float]]], check: bool = True) -> Obj2d`
Create a polygon from a single or multiple closed paths of points.

`rectangle(size: list[float, float], center: bool = False) -> Obj2d`
Make a rectangle of a given size.

`rounded_rectangle(size: list[float, float], rounding_radius: float = 0.2, segments: int = -1, center: bool = False) -> Obj2d`
Create a rectangle with rounded corners.

`square(size: float, center: bool = False) -> Obj2d`
Make a square of a given size.

`star(num_points: int, outer_radius: float, inner_radius: float = 0.0) -> Obj2d`
Make a regular star of a given number of points.

`text(size: float, text: str, inter_char_space=None) -> Obj2d`
Draw the unicode printable characters in `text` in shapes of size `size`.

```
text_set_font(font_name: str) -> None
```
Set the current font used by `text`.

3d Primitives

```
cone(height: float, radius_low: float,
radius_high: float = 0.2, segments: int = -1,
center: bool = False) -> Obj3d
```
Make a cone with given radii and height.

```
cube(size: float, center: bool = False) -> Obj3d
```
Make a cube of with sides of the given size.

```
cuboid(size: list[float, float, float], center:
bool = False) -> Obj3d
```
Make a cuboid with the x, y, and z values given in size.

```
cylinder(height: float, radius: float, segments:
int = -1, center=False) -> Obj3d
```
Make a cylinder of a given radius and height.

```
ellipsoid(radii: tuple[float, float, float],
segments: int = -1, center=False) -> Obj3d
```
Make an ellipsoid which is an elliptical on all three radii.

```
elliptical_cylinder(height: float, radii:
tuple[float, float], segments: int = -1,
center=False) -> Obj3d
```
Make a elliptically shaped cylinder of given radii and height.

```
extrude(obj: Obj2d, height: float) -> Obj3d
```
Create a Obj3d solid from Obj2d of given height.

```
extrude_chaining(pairs: list[tuple[float, Obj2d]],
is_convex: bool = False, diagnose: str = None) ->
Obj3d
```
Extrude multiple 2d objects into a single 3d Object.

```
extrude_transforming(obj: Obj2d, height: float,
num_twist_divisions: int = 0, twist: float = 0,
scale: tuple[float, float] = (1.0, 1.0)) -> Obj3d
```
Create a Obj3d solid from Obj2d of given height.

```
geodesic_sphere(radius, segments=-1) -> Obj3d
```
Create a geodesic sphere of a given radius.

```
lithophane(image_filename: str, width_mm: int =
150, pixel_size: float = 0.5, min_thickness: float
= 0.8, max_thickness: float = 3.0) -> Obj3d
```
Create a 3d lithophane from a 2d image.

```
polyhedron(vertices: list[tuple[float, float,
float]], faces: list[tuple[int, int, int]], check:
str = 'interactive') -> Obj3d
```
Create an Obj3d from points and a list of triangles using those points.

```
pyramid(height: int, num_sides: int, radius:
float) -> Obj3d
```
Make a regular pyramid with the given height and number of sides.

```
revolve(obj: Obj2d, revolve_degrees: float =
360.0, segments: int = -1) -> Obj3d
```
Create a Obj3d by revolving an Obj2d around the Y-axis, then rotating it so that Y becomes Z.

```
rounded_cuboid(size: list[float, float, float],
rounding_radius=2.0, segments: int = -1, center:
bool = False) -> Obj3d
```
Make a rounded_cuboid with the x, y, and z values given in size.

```
rounded_cylinder(height: float, radius: float,
rounding_radius: float = 2.0, segments: int = -1,
center: bool = False) -> Obj3d
```
Make a rounded cylinder of a given radius and height.

`sphere(radius: float, segments: int = -1) -> Obj3d`
Create a classical sphere of a given radius.

`tetrahedron(vertices: list[tuple[float, float, float]] | float | int = None) -> Obj3d`
Create a tetrahedron from 4 `vertices`.

`torus(outer_radius: float, inner_radius: float, segments=-1) -> Obj3d`
Create a torus with the specified radii.

Trigonometry (degrees)

`acos(cosVal: float) -> float`
ArcCosine of cosVal returning angle in degrees.

`acosh(cosVal: float) -> float`
ArcCosine of hyperbolic cosVal returning angle in degrees.

`asin(sinVal: float) -> float`
ArcSine of sinVal returning angle in degrees.

`asinh(sinVal: float) -> float`
ArcSine of hyperbolic sinVal returning angle in degrees

`atan(tanVal: float) -> float`
ArcTan of tanVal returning angle in degrees.

`atan2(y: float, x: float) -> float`
ArcTan of quotient of y and x returning angle in degrees.

`atanh(tanVal: float) -> float`
ArcTan of hyperbolic tanVal returning angle in degrees.

`cos(angleInDegrees: float) -> float`
Cosine of angle in degrees.

`cosh(angleInDegrees: float) -> float`
Hyperbolic cosine of angle in degrees.

`deg_to_rad(angleInDegrees: float) -> float`
Convert degrees to radians.

`rad_to_deg(angleInRadians: float) -> float`
Convert radians to degrees.

`sin(angleInDegrees: float) -> float`
Sine of angle in degrees.

`sinh(angleInDegrees: float) -> float`
Hyperbolic sine of angle in degrees.

`tan(angleInDegrees: float) -> float`
Tangent of angle in degrees.

`tanh(angleInDegrees: float) -> float`
Hyperbolic tangent of angle in degrees.

Utility Functions

`check_mesh(vertices: list[tuple[float, float, float]], faces: list[tuple[int, int, int]]) -> bool`
Check manifold and winding of the mesh defined in vertices and faces.

`load(filename: str) -> Obj3d | Obj2d`
Load a 3d object from a file.

`quick_check_mesh(vertices: list[tuple[float, float, float]], faces: list[tuple[int, int, int]]) -> str`
Same checking as check_mesh, but no advice is offered

`save(filename: str, *objs: Obj3d | Obj2d) -> None`
Save a 3d or 2d object in a file suitable for printing, etc.

`view(obj: Obj3d | Obj2d, title: str = '') -> None`
Use `Piecad-Viewer` to display the geometry object.

`winding(lt: list[tuple[float, float]]) -> str`
String description of winding of a 2D polygon.

class Obj2d

`area(self) -> 'float'`
The area of this Obj2d.

`bounding_box(self) -> 'tuple[float, float, float, float]'`
Return the bounding box of this object.

`center(self, axes: 'tuple[bool, bool]' = (True, True), at: 'tuple[float, float]' = (0, 0)) -> 'Obj2d'`
Center the object on each of the `True` axes.

`color(self, cspec: 'tuple[int, int, int] | str') -> 'Obj2d'`
Assign the given color to this object.

`corner(self, at: 'tuple[float, float]' = (0, 0)) -> 'Obj2d'`
Move the bounding box minimum corner to (0, 0).

`decompose(self) -> 'list[Obj2d]'`
Decompose this object into a list of topologically disjoint objects.

`extrude(self, height: 'int') -> 'Obj3d'`
Extrude this object into a Obj3d of the given height.

`is_empty(self) -> 'bool'`
Is this object empty?

`mirror(self, axes: 'tuple[bool, bool]') -> 'Obj2d'`
Mirror this object around the given axes.

`num_verts(self) -> 'int'`
The number of vertices in this object.

`offset(self, delta: 'float', join_type: 'str', miter_limit: 'float' = 2.0, segments: 'int' = -1) -> 'Obj2d'`
Offset (or inset) a 2D object by a given distance called `delta`.

`piecut(self, start_angle=0, end_angle=90, both=False) -> 'Obj2d | tuple[Obj2d, Obj2d]'`
Cut a wedge out of this object.

`resize(self, sizes: 'list[tuple[float | None, float | None]]') -> 'Obj2d'`
Resize this object to a specific size on each axis.

`revolve(self, revolve_degrees: 'float' = 360.0, segments: 'int' = -1) -> 'Obj3d'`
Create a Obj3d by revolving this object around the Y-axis, then rotating it so that Y becomes Z.

`rotate(self, degrees: 'float') -> 'Obj2d'`
Rotate this object by the given degrees.

`scale(self, factors: 'list[float, float]') -> 'Obj2d'`
Scale this object by the given factors.

`to_paths(self) -> 'list[list[float, float]]'`
Return a lists of paths, each of which is a list of vertices that make up this object.

```
transform(self, matrix2x3: 'tuple[tuple[float, float, float], tuple[float, float, float]]') -> 'Obj2d'
```
Transform this object with the given affine transformaton matrix.

```
translate(self, offsets: 'list[float, float]') -> 'Obj2d'
```
Translate (move) this object by the given offsets.

class Obj3d

```
bounding_box(self) -> 'tuple[float, float, float, float, float, float]'
```
Return the bounding box of this object.

```
center(self, axes: 'tuple[bool, bool, bool]' = (True, True, True), at: 'tuple[float, float, float]' = (0, 0, 0)) -> 'Obj3d'
```
Center the object on each of the `True` axes.

```
color(self, cspec: 'tuple[int, int, int] | str') -> 'Obj3d'
```
Assign the given color to this object.

```
corner(self, at: 'tuple[float, float, float]' = (0, 0, 0)) -> 'Obj3d'
```
Move the bounding box minimum corner to (0, 0, 0).

```
decompose(self) -> 'list[Obj3d]'
```
Decompose this object into a list of topologically disjoint objects.

```
is_empty(self) -> 'bool'
```
Is this object empty?

```
minkowski_difference(self, other: 'Obj3d') -> 'Obj3d'
```
Return the minkowski_sum of this object and other.

```
minkowski_sum(self, other: 'Obj3d') -> 'Obj3d'
```
Return the minkowski_sum of this object and other.

```
mirror(self, axes: 'tuple[bool, bool, bool]') -> 'Obj3d'
```
Mirror this object around the given axes.

```
miter_cut(self, cut_angle: 'float', cut_point: 'tuple[float, float, float]') -> 'tuple[Obj3d, Obj3d]'
```
Cut this object into two parts using a plane defined by the `cut_angle` and `cut_point`.

```
num_faces(self) -> 'int'
```
The number of faces in this object.

```
num_verts(self) -> 'int'
```
The number of vertices in this object.

```
piecut(self, start_angle=0, end_angle=90, both=False) -> 'Obj3d | tuple[Obj3d, Obj3d]'
```
Cut a wedge out of this object.

```
project(self) -> 'Obj2d'
```
Return a Obj2d representing this object's "shadow" on the x-y plane.

```
resize(self, sizes: 'list[tuple[float | None, float | None, float | None]]') -> 'Obj3d'
```
Resize this object to a specific size on each axis.

```
rotate(self, degrees: 'list[float, float, float]') -> 'Obj3d'
```
Rotate this object by the given degrees for each axis.

```
scale(self, factors: 'list[float, float, float]') -> 'Obj3d'
```
Scale this object by the given factors.

`slice(self, height: 'float') -> 'Obj2d'`
Like `project`, but rather than the bottom, project at the given height.

`split(self, cutter: 'Obj3d') -> 'Obj3d'`
This more efficently does a difference and an intersect operation between this and cutter.

`surface_area(self) -> 'float'`
The surface area of this Obj3d.

```
to_verts_and_faces(self) ->
'tuple[list[list[float, float, float]],
list[list[int, int, int]]]'
```
Return a pair containg a list of vertices and a list of faces for this object.

```
transform(self, matrix3x4: 'tuple[tuple[float,
float, float, float], tuple[float, float, float,
float], tuple[float, float, float, float],
tuple[float, float, float, float]]') -> 'Obj3d'
```
Transform this object with the given affine transformaton matrix.

```
translate(self, offsets: 'list[float, float,
float]') -> 'Obj3d'
```
Translate (move) this object by the given offsets.

`volume(self) -> 'float'`
The volume of this Obj3d.

class Config

```
Config.get_default_color() -> 'tuple[int, int,
int]'
```
Returns the RGB values for the current default color.

`Config.get_default_segments() -> 'int'`
Get the default number of segments used by circular objects.

`Config.get_default_units() -> 'str'`
Get the default units used by this script.

`Config.get_layer_resolution() -> 'float'`
Get the layer resolution used in printing in this script.

```
Config.set_default_color(cspec: 'tuple[int, int,
int] | str') -> 'None'
```
Assign the given color for use as a default color for objects.

```
Config.set_default_segments(segments: 'int' = 36)
-> 'None'
```
Set the default number of segments used by circular objects.

```
Config.set_default_units(units: 'str' = 'mm') ->
'None'
```
Set the default units used in this script.

```
Config.set_layer_resolution(resolution: 'float') -
> 'None'
```
Set the layer resolution used in printing in this script.