File - C:\GitRepos\geomeppy\geomeppy\builder.py

```python
"""Build IDF geometry from minimal inputs."""
from typing import Any, Dict, List, Tuple, Union  # noqa

from six.moves import zip

from .geom.polygons import Polygon, Polygon3D  # noqa
from .geom.segments import Segment  # noqa
from .geom.vectors import Vector3D


class Zone(object):
    """Represents a single zone for translation into an IDF."""

    def __init__(self, name, surfaces):
        # type: (str, Dict[str, Any]) -> None
        """Initialise the zone object.

        :param name: A name for the zone.
        :param surfaces: The surfaces that make up the zone.

        """
        self.name = name
        self.walls = [s for s in surfaces["walls"] if s.area > 0]
        self.floors = surfaces["floors"]
        self.roofs = surfaces["roofs"]
        self.ceilings = surfaces["ceilings"]


class Block(object):
    def __init__(
        self,
        name,  # type: str
        coordinates,  # type: Union[List[Tuple[float, float]], List[Tuple[int, int]]]
        height,  # type: float
        num_stories=1,  # type: int
        below_ground_stories=0,  # type: int
        below_ground_storey_height=2.5,  # type: float
        zoning="by_storey",  # type: str
        perim_depth=3.0,  # type: float
    ):
        # type: (...) -> None
        """Represents a single block for translation into an IDF.

        :param name: A name for the block.
        :param coordinates: A list of (x, y) tuples representing the building outline.
        :param height: The height of the block roof above ground level.
        :param num_stories: The total number of stories including basement stories. Default : 1.
        :param below_ground_stories: The number of stories below ground. Default : 0.
        :param below_ground_storey_height: The height of each basement storey. Default : 2.5.
        :param zoning: The zoning pattern of the block. Default : by_storey
        :param perim_depth: Depth of the perimeter zones if the core/perim zoning pattern is requested. Default : 3.0.

        """
        self.name = name
        if coordinates[0] == coordinates[-1]:
            coordinates.pop()
        self.coordinates = coordinates
        self.height = height
        self.num_stories = num_stories
        self.num_below_ground_stories = below_ground_stories
        self.below_ground_storey_height = below_ground_storey_height
        self.zoning = zoning
        self.perim_depth = perim_depth

    @property
    def stories(self):
        # type: () -> List[Dict[str, Any]]
        """A list of dicts of the surfaces of each storey in the block.

        :returns: list of dicts

        Example dict format::
            {'floors': [...],
             'ceilings': [...],
             'walls': [...],
             'roofs': [...],
             }

        """
        stories = []
        if self.num_below_ground_stories != 0:
            floor_no = -self.num_below_ground_stories
        else:
            floor_no = 0
        for floor, ceiling, wall, roof in zip(
            self.floors, self.ceilings, self.walls, self.roofs
        ):
            stories.append(
                {
                    "storey_no": floor_no,
                    "floors": floor,
                    "ceilings": ceiling,
                    "walls": wall,
                    "roofs": roof,
```

Just a DTO including all surfaces of a Zone.

storey_no ← for naming in IDF.add_block.

Dict with keys: floors : [0], ceilings : [0], walls : [0], roofs : [0] — generated by corresponding getters in Block.

each element is List[Polygon].

instantiation in IDF.add_block.
LOC 264, 282.

each dict contains surfaces of different types (as keys of dict)

```
95                  }
96              )
97              floor_no += 1
98          return stories
99
100     @property
101     def footprint(self):
102         # type: () -> Polygon3D
103         """Ground level outline of the block.
104
105         :returns: A 2D outline of the block.
106
107         """
108         coordinates = [(v[0], v[1], 0) for v in self.coordinates]
109         return Polygon3D(coordinates)
110
111     @property
112     def storey_height(self):
113         # type: () -> float
114         """Height of above ground stories.
115
116         :returns: Average storey height.
117
118         """
119         return float(self.height) / (self.num_stories - self.num_below_ground_stories)
120
121     @property
122     def floor_heights(self):
123         # type: () -> list[float]
124         """Floor height for each storey in the block.
125
126         :returns: A list of floor heights.
127
128         """
129         lfl = self.lowest_floor_level
130         sh = self.storey_height
131         floor_heights = [lfl + sh * i for i in range(self.num_stories)]
132         return floor_heights
133
134     @property
135     def ceiling_heights(self):
136         # type: () -> list[float]
137         """Ceiling height for each storey in the block.
138
139         :returns: A list of ceiling heights.
140
141         """
142         lfl = self.lowest_floor_level
143         sh = self.storey_height
144         ceiling_heights = [lfl + sh * (i + 1) for i in range(self.num_stories)]
145         return ceiling_heights
146
147     @property
148     def lowest_floor_level(self):
149         # type: () -> float
150         """Floor level of the lowest basement storey.
151
152         :returns: Lowest floor height.
153
154         """
155         return -(self.num_below_ground_stories * self.below_ground_storey_height)
156
157     @property
158     def walls(self):
159         # type: () -> list[list[Polygon3D]]
160         """Coordinates for each wall in the block.
161
162         These are ordered as a list of lists, one for each storey.
163
164         :returns: Coordinates for all walls.
165
166         """
167         walls = []
168         for fh, ch in zip(self.floor_heights, self.ceiling_heights):
169             floor_walls = [_make_wall(edge, fh, ch) for edge in self.footprint.edges]
170             walls.append(floor_walls)
171         return walls
172
173     @property
174     def floors(self):
175         """Coordinates for each floor in the block.
176
177         :returns: Coordinates for all floors.
178
179         """
180         floors = [
181             [self.footprint.invert_orientation() + Vector3D(0, 0, fh)]
182             for fh in self.floor_heights
183         ]
184         return floors
185
186     @property
187     def ceilings(self):
188         """Coordinates for each ceiling in the block.
```

Turn 2D coord. List to 3D.

turn 3D triple coord. to Polygon3D obj.

↳ has superclass { Polygon / Clipper3D.

__init__ turns coord. to self. vertices.

seems to be accurate only when above and below ground storeys all have the same height.

Polygon.edges: List of Segments.

LOC 231. Polygon3D constructed with 4 vertices.

List² of Polygon3Ds. → one per edge x floor

Polygon.invert_orientation(): reverse the order of the vertices.

List² of Polygon3Ds for all flurs at different heights.

```
189
190         :returns: Coordinates for all ceilings.
191
192         """
193         ceilings = [
194             [self.footprint + Vector3D(0, 0, ch)] for ch in self.ceiling_heights[:-1]
195         ]
196
197         ceilings.append([])
198         return ceilings
199
200     @property
201     def roofs(self):
202         """Coordinates for each roof of the block.
203
204         This returns a list with an entry for each floor for consistency with
205         the other properties of the Block object, but should only have roof
206         coordinates in the list in the final position.
207
208         :returns: Coordinates for all roofs.
209
210         """
211         roofs = [[] for ch in self.ceiling_heights[:-1]]  # type: List[List[Polygon]]
212         roofs.append([self.footprint + Vector3D(0, 0, self.height)])
213         return roofs
214
215     @property
216     def surfaces(self):
217         # type: () -> Dict[str, Any]
218         """Coordinates for all the surfaces in the block.
219
220         :returns: Coordinates for all surfaces.
221
222         """
223         return {
224             "walls": self.walls,
225             "ceilings": self.ceilings,
226             "roofs": self.roofs,
227             "floors": self.floors,
228         }
229
230
231 def _make_wall(edge, floor_height, ceiling_height):
232     # type: (Segment, float, float) -> Polygon3D
233     """Create a polygon representing the vertices of a wall.
234
235     :param edge: Segment of a floor outline at ground level.
236     :param floor_height: Floor height.
237     :param ceiling_height: Ceiling height.
238
239     """
240     return Polygon3D(
241         [
242             edge.p1 + (0, 0, ceiling_height),  # upper left
243             edge.p1 + (0, 0, floor_height),  # lower left
244             edge.p2 + (0, 0, floor_height),  # lower right
245             edge.p2 + (0, 0, ceiling_height),  # upper right
246         ]
247     )
248
```

→ take out roof level.

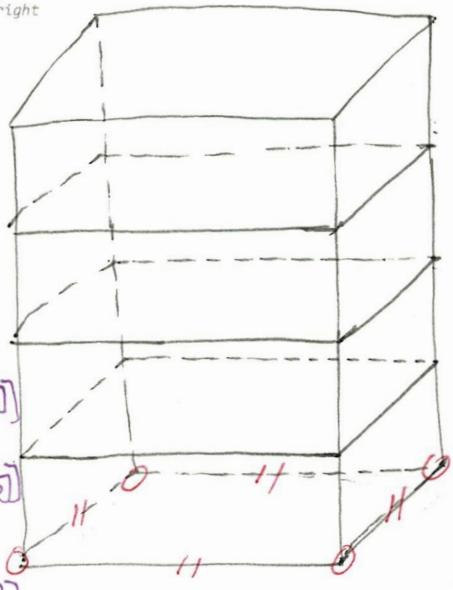
— 1 empty []

→ (#fl - 1) empty []

getter of all surfaces. :-\

Only used by unit tests.

helper for self.walls.



File - C:\GitRepos\geomeppy\geomeppy\idf.py

```python
"""
This module contains the implementation of `geomeppy.IDF`.
"""
import itertools
from typing import Any, Dict, List, Optional, Union  # noqa

from eppy.bunch_subclass import EpBunch  # noqa
from eppy.idf_msequence import Idf_MSequence  # noqa

from .geom.intersect_match import intersect_idf_surfaces, match_idf_surfaces
from .builder import Block, Zone
from .geom.polygons import bounding_box, Polygon2D  # noqa
from .geom.vectors import Vector2D, Vector3D  # noqa
from .io.obj import export_to_obj
from .patches import PatchedIDF
from .recipes import (
    set_default_constructions,
    set_wwr,
    rotate,
    scale,
    translate,
    translate_to_origin,
)
from .view_geometry import view_idf
from .geom.core_perim import core_perim_zone_coordinates


def new_idf(fname):
    # type: (str) -> IDF
    """Create a new blank IDF.

    :param fname: A name for the new IDF.

    """
    idf = IDF()
    idf.new(fname)
    return idf


class IDF(PatchedIDF):
    """Geometry-enabled IDF class, usable in the same way as Eppy's IDF.

    This adds geometry functionality to Eppy's IDF class.

    """

    def intersect_match(self):
        # type: () -> None
        """Intersect all surfaces in the IDF, then set boundary conditions."""
        self.intersect()
        self.match()

    def intersect(self):
        # type: () -> None
        """Intersect all surfaces in the IDF."""
        intersect_idf_surfaces(self)

    def match(self):
        # type: () -> None
        """Set boundary conditions for all surfaces in the IDF."""
        match_idf_surfaces(self)

    def translate_to_origin(self):
        # type: () -> None
        """Move an IDF close to the origin so that it can be viewed in SketchUp."""
        translate_to_origin(self)

    def translate(self, vector):
        # type: (Vector2D) -> None
        """Move the IDF in the direction given by a vector.

        :param vector: A vector to translate by.

        """
        surfaces = self.getsurfaces()
        translate(surfaces, vector)
        subsurfaces = self.getsubsurfaces()
        translate(subsurfaces, vector)
        shadingsurfaces = self.getshadingsurfaces()
        translate(shadingsurfaces, vector)

    def rotate(self, angle, anchor=None):
        # type: (float, Optional[Union[Vector2D, Vector3D]]) -> None
        """Rotate the IDF counterclockwise by the angle given.

        :param angle: Angle (in degrees) to rotate by.
        :param anchor: Point around which to rotate. Default is the centre of the the IDF's bounding box.

        """
        anchor = anchor or self.centroid
        surfaces = self.getsurfaces()
        subsurfaces = self.getsubsurfaces()
        shadingsurfaces = self.getshadingsurfaces()
        self.translate(-anchor)
```

geom/intersect_match.py.

recipes.py

recipes.py

recipes.py.

File - C:\GitRepos\geomeppy\geomeppy\idf.py

```
 95          rotate(surfaces, angle)
 96          rotate(subsurfaces, angle)
 97          rotate(shadingsurfaces, angle)
 98          self.translate(anchor)
 99
100      def scale(self, factor, anchor=None, axes="xy"):
101          # type: (float, Optional[Union[Vector2D, Vector3D]], str) -> None
102          """Scale the IDF by a scaling factor.
103
104          :param factor: Factor to scale by.
105          :param anchor: Point to scale around. Default is the centre of the the IDF's bounding box.
106          :param axes: Axes to scale on. Default 'xy'.
107
108          """
109          anchor = anchor or self.centroid
110          surfaces = self.getsurfaces()
111          subsurfaces = self.getsubsurfaces()
112          shadingsurfaces = self.getshadingsurfaces()
113          self.translate(-anchor)
114          scale(surfaces, factor, axes)
115          scale(subsurfaces, factor, axes)
116          scale(shadingsurfaces, factor, axes)
117          self.translate(anchor)
118
119      def set_default_constructions(self):
120          # type: () -> None
121          set_default_constructions(self)
122
123      def bounding_box(self):
124          # type: () -> Polygon2D
125          """Calculate the site bounding box.
126
127          :returns: A polygon of the bounding box.
128
129          """
130          floors = self.getsurfaces("floor")
131          return bounding_box(floors)
132
133      @property
134      def centroid(self):
135          # type: () -> Vector2D
136          """Calculate the centroid of the site bounding box.
137
138          :returns: The centroid of the site bounding box.
139
140          """
141          bbox = self.bounding_box()
142          return bbox.centroid
143
144      def getsurfaces(self, surface_type=""):
145          # type: (str) -> Union[list[EpBunch], Idf_MSequence]
146          """Return all surfaces in the IDF.
147
148          :param surface_type: Type of surface to get. Defaults to all.
149          :returns: IDF surfaces.
150
151          """
152          surfaces = itertools.chain.from_iterable(
153              [
154                  self.idfobjects[key.upper()]
155                  for key in self.idd_index["ref2names"]["SurfaceNames"]
156              ]
157          )
158          if surface_type:
159              surfaces = filter(
160                  lambda x: x.Surface_Type.lower() == surface_type.lower(), surfaces
161              )
162          return list(surfaces)
163
164      def getsubsurfaces(self, surface_type=""):
165          # type: (str) -> Union[list[EpBunch], Idf_MSequence]
166          """Return all subsurfaces in the IDF.
167
168          :param surface_type: Type of surface to get. Defaults to all.
169          :returns: IDF surfaces.
170
171          """
172          surfaces = itertools.chain.from_iterable(
173              [
174                  self.idfobjects[key.upper()]
175                  for key in self.idd_index["ref2names"]["SubSurfNames"]
176              ]
177          )
178          if surface_type:
179              surfaces = filter(
180                  lambda x: x.Surface_Type.lower() == surface_type.lower(), surfaces
181              )
182          return list(surfaces)
183
184      def getshadingsurfaces(self, surface_type=""):
185          # type: (str) -> Union[list[EpBunch], Idf_MSequence]
186          """Return all subsurfaces in the IDF.
187
188          :param surface_type: Type of surface to get. Defaults to all.
```

recipes.py.

recipes.py. LOC23-95.

Hard-coded.
cons name in Title Case
mat name in Upper Camel Case

File - C:\GitRepos\geomeppy\geomeppy\idf.py

```
189         :returns: IDF surfaces.
190
191         """
192         surfaces = itertools.chain.from_iterable(
193             [
194                 self.idfobjects[key.upper()]
195                 for key in self.idd_index["ref2names"]["AllShadingSurfNames"]
196             ]
197         )
198         if surface_type:
199             surfaces = filter(
200                 lambda x: x.Surface_Type.lower() == surface_type.lower(), surfaces
201             )
202         return list(surfaces)
203
204     def set_wwr(
205         self, wwr=0.2, construction=None, force=False, wwr_map={}, orientation=None
206     ):
207         # type: (Optional[float], Optional[str], Optional[bool], Optional[dict], Optional[str]) -> None
208         """Add strip windows to all external walls.
209
210         Different WWR can be applied to specific wall orientations using the `wwr_map` keyword arg.
211         This map is a dict of wwr values, keyed by `wall.azimuth`, which overrides the default passed as `wwr`.
212
213         They can also be applied to walls oriented to a compass point, e.g. north, which will apply to walls which
214         have an azimuth within 45 degrees of due north.
215
216         :param wwr: Window to wall ratio in the range 0.0 to 1.0.
217         :param construction: Name of a window construction.
218         :param force: True to remove all subsurfaces before setting the WWR.
219         :param wwr_map: Mapping from wall orientation (azimuth) to WWR, e.g. {180: 0.25, 90: 0.2}.
220         :param orientation: One of "north", "east", "south", "west". Walls within 45 degrees will be affected.
221
222         """
223         set_wwr(self, wwr, construction, force, wwr_map, orientation)
224
225     def view_model(self, test=False):
226         # type: (Optional[bool]) -> None
227         """Show a zoomable, rotatable representation of the IDF."""
228         view_idf(idf=self, test=test)
229
230     def to_obj(self, fname=None, mtllib=None):
231         # type: (Optional[str], Optional[str]) -> None
232         """Export an OBJ file representation of the IDF.
233
234         This can be used for viewing in tools which support the .obj format.
235
236         :param fname: A filename for the .obj file. If None we try to base it on IDF.idfname and change the filetype.
237         :param mtllib: The name of a .mtl file to be referenced from the .obj file. If None, we use default.mtl.
238         """
239         if not fname:
240             try:
241                 fname = self.idfname.replace(".idf", ".obj")
242             except AttributeError:
243                 fname = "default.obj"
244         export_to_obj(self, fname, mtllib)
245
246     def add_block(self, *args, **kwargs):
247         # type: (*Any, **Any) -> None
248         """Add a block to the IDF.
249
250         :param name: A name for the block.
251         :param coordinates: A list of (x, y) tuples representing the building outline.
252         :param height: The height of the block roof above ground level.
253         :param num_stories: The total number of stories including basement stories. Default : 1.
254         :param below_ground_stories: The number of stories below ground. Default : 0.
255         :param below_ground_storey_height: The height of each basement storey. Default : 2.5.
256         :param zoning: The zoning pattern of the block. Default : by_storey
257         :param perim_depth: Depth of the perimeter zones if the core/perim zoning pattern is requested. Default : 3.0.
258
259         """
260         block = Block(*args, **kwargs)
261         block.zoning = kwargs.get("zoning", "by_storey")
262         if block.zoning == "by_storey":
263             zones = [
264                 Zone("Block %s Storey %i" % (block.name, storey["storey_no"]), storey)
265                 for storey in block.stories
266             ]
267         elif block.zoning == "core/perim":
268             zones = []
269             try:
270                 for name, coords in core_perim_zone_coordinates(
271                     block.coordinates, block.perim_depth
272                 )[0].iteritems():
273                     block = Block(
274                         name=name,
275                         coordinates=coords,
276                         height=block.height,
277                         num_stories=block.num_stories,
278                     )
279                     zones += [
280                         Zone(
281                             "Block %s Storey %i" % (block.name, storey["storey_no"]),
282                             storey,
```

Page 3 of 4

File - C:\GitRepos\geomeppy\geomeppy\idf.py

```
283                     )
284                     for storey in block.stories
285                 ]
286             except NotImplementedError:
287                 raise ValueError("Perimeter depth is too great")
288         else:
289             raise ValueError("%s is not a valid zoning rule" % block.zoning)
290
291         for zone in zones:
292             self.add_zone(zone)
293
294     def add_shading_block(self, *args, **kwargs):
295         # type: (*Any, **Any) -> None
296         """Add a shading block to the IDF.
297
298         :param name: A name for the block.
299         :param coordinates: A list of (x, y) tuples representing the building outline.
300         :param height: The height of the block roof above ground level.
301         :param num_stories: The total number of stories including basement stories. Default : 1.
302         :param below_ground_stories: The number of stories below ground. Default : 0.
303         :param below_ground_storey_height: The height of each basement storey. Default : 2.5.
304
305         """
306         block = Block(*args, **kwargs)
307         for i, wall in enumerate(block.walls[0], 1):
308             if wall.area <= 0:
309                 continue
310             s = self.newidfobject(
311                 "SHADING:SITE:DETAILED", Name="%s_%s" % (block.name, i)
312             )
313             try:
314                 s.setcoords(wall)
315             except ZeroDivisionError:
316                 self.removeidfobject(s)
317
318     def add_zone(self, zone):
319         # type: (Zone) -> None
320         """Add a zone to the IDF.
321
322         :param zone: A Zone object holding details about the zone.
323
324         """
325         try:
326             ggr = self.idfobjects["GLOBALGEOMETRYRULES"][
327                 0
328             ]  # type: Optional[Dict[str, Idf_MSequence]]
329         except IndexError:
330             ggr = None
331         # add zone object
332         self.newidfobject("ZONE", Name=zone.name)
333
334         for surface_type in zone.__dict__.keys():
335             if surface_type == "name":
336                 continue
337             for i, surface_coords in enumerate(zone.__dict__[surface_type], 1):
338                 if not surface_coords:
339                     continue
340                 name = "{name} {s_type} {num:04d}".format(
341                     name=zone.name, s_type=surface_type[:-1].title(), num=i
342                 )
343                 s = self.newidfobject(
344                     "BUILDINGSURFACE:DETAILED",
345                     Name=name,
346                     Surface_Type=surface_type[:-1],
347                     Zone_Name=zone.name,
348                 )
349                 s.setcoords(surface_coords, ggr)
350
```

only usage of add_zone Loc 318

Ideas: Add.

add_single_zone → only add one zone with args similar to add_block.

'Block' is a non-physical concept in Geomeppy.

We can do walk around for now. no potential dangers found.

File - C:\GitRepos\geomeppy\geomeppy\recipes.py

```python
"""Recipes for making changes to EnergyPlus IDF files.

These are generally exposed and methods on the IDF object, e.g. `set_default_constructions(idf)`
can be called on an existing `IDF` object like ``myidf.set_default_constructions()``.

"""
from typing import List, Optional, Tuple, Union  # noqa
import warnings

from eppy.idf_msequence import Idf_MSequence  # noqa
import numpy as np

from .geom.polygons import Polygon3D
from .geom.transformations import Transformation
from .geom.vectors import Vector2D, Vector3D  # noqa

if False:
    from .idf import IDF  # noqa
if False:
    from .patches import EpBunch  # noqa


def set_default_constructions(idf):
    # type: (IDF) -> None
    """Set default constructions for surfaces in the model.

    :param idf: The IDF object.

    """
    constructions = [
        "Project Wall",
        "Project Partition",
        "Project Floor",
        "Project Flat Roof",
        "Project Ceiling",
        "Project Door",
    ]
    for construction in constructions:
        idf.newidfobject(
            "CONSTRUCTION", Name=construction, Outside_Layer="DefaultMaterial"
        )
    idf.newidfobject(
        "MATERIAL",
        Name="DefaultMaterial",
        Roughness="Rough",
        Thickness=0.1,
        Conductivity=0.1,
        Density=1000,
        Specific_Heat=1000,
    )

    idf.newidfobject(
        "CONSTRUCTION", Name="Project External Window", Outside_Layer="DefaultGlazing"
    )
    idf.newidfobject(
        "WINDOWMATERIAL:SIMPLEGLAZINGSYSTEM",
        Name="DefaultGlazing",
        UFactor=2.7,
        Solar_Heat_Gain_Coefficient=0.763,
        Visible_Transmittance=0.8,
    )

    for surface in idf.getsurfaces():
        set_default_construction(surface)
    for subsurface in idf.getsubsurfaces():
        set_default_construction(subsurface)


def set_default_construction(surface):
    # type: (EpBunch) -> None
    """Set default construction for a surface in the model.

    :param surface: A model surface.

    """
    if surface.Surface_Type.lower() == "wall":
        if surface.Outside_Boundary_Condition.lower() == "outdoors":
            surface.Construction_Name = "Project Wall"
        elif surface.Outside_Boundary_Condition.lower() == "ground":
            surface.Construction_Name = "Project Wall"
        else:
            surface.Construction_Name = "Project Partition"
    if surface.Surface_Type.lower() == "floor":
        if surface.Outside_Boundary_Condition.lower() == "ground":
            surface.Construction_Name = "Project Floor"
        else:
            surface.Construction_Name = "Project Floor"
    if surface.Surface_Type.lower() == "roof":
        surface.Construction_Name = "Project Flat Roof"
    if surface.Surface_Type.lower() == "ceiling":
        surface.Construction_Name = "Project Ceiling"
    if surface.Surface_Type == "window":
        surface.Construction_Name = "Project External Window"
    if surface.Surface_Type == "door":
```

```python
        surface.Construction_Name = "Project Door"


def set_wwr(
    idf, wwr=0.2, construction=None, force=False, wwr_map=None, orientation=None
):
    # type: (IDF, Optional[float], Optional[str], Optional[bool], Optional[dict], Optional[str]) -> None
    """Set the window to wall ratio on all external walls.

    :param idf: The IDF to edit.
    :param wwr: The window to wall ratio.
    :param construction: Name of a window construction.
    :param force: True to remove all subsurfaces before setting the WWR.
    :param wwr_map: Mapping from wall orientation (azimuth) to WWR, e.g. {180: 0.25, 90: 0.2}.
    :param orientation: One of "north", "east", "south", "west". Walls within 45 degrees will be affected.

    """
    try:
        ggr = idf.idfobjects["GLOBALGEOMETRYRULES"][0]  # type: Optional[Idf_MSequence]
    except IndexError:
        ggr = None

    # check orientation
    orientations = {
        "north": 0.0,
        "east": 90.0,
        "south": 180.0,
        "west": 270.0,
        None: None,
    }
    degrees = orientations.get(orientation, None)
    external_walls = filter(
        lambda x: x.Outside_Boundary_Condition.lower() == "outdoors",
        idf.getsurfaces("wall"),
    )
    external_walls = filter(
        lambda x: _has_correct_orientation(x, degrees), external_walls
    )
    subsurfaces = idf.getsubsurfaces()
    base_wwr = wwr
    for wall in external_walls:
        # get any subsurfaces on the wall
        wall_subsurfaces = list(
            filter(lambda x: x.Building_Surface_Name == wall.Name, subsurfaces)
        )
        if not all(_is_window(wss) for wss in wall_subsurfaces) and not force:
            raise ValueError(
                'Not all subsurfaces on wall "{name}" are windows. '
                "Use `force=True` to replace all subsurfaces.".format(name=wall.Name)
            )

        if wall_subsurfaces and not construction:
            constructions = list(
                {wss.Construction_Name for wss in wall_subsurfaces if _is_window(wss)}
            )
            if len(constructions) > 1:
                raise ValueError(
                    'Not all subsurfaces on wall "{name}" have the same construction'.format(
                        name=wall.Name
                    )
                )
            construction = constructions[0]
        # remove all subsurfaces
        for ss in wall_subsurfaces:
            idf.removeidfobject(ss)
        wwr = (wwr_map or {}).get(wall.azimuth) or base_wwr
        if not wwr:
            return
        coords = window_vertices_given_wall(wall, wwr)
        window = idf.newidfobject(
            "FENESTRATIONSURFACE:DETAILED",
            Name="%s window" % wall.Name,
            Surface_Type="Window",
            Construction_Name=construction or "",
            Building_Surface_Name=wall.Name,
            View_Factor_to_Ground="autocalculate",  # from the surface angle
        )
        window.setcoords(coords, ggr)


def _has_correct_orientation(wall, orientation_degrees):
    # type: (EpBunch, Optional[float]) -> bool
    """Check that the wall has an orientation which requires WWR to be set.

    :param wall: An EpBunch representing a wall.
    :param orientation_degrees: Orientation in degrees.
    :return: True if the wall is within 45 degrees of the orientation passed, or no orientation passed.
             False if the wall is not within 45 of the orientation passed.
    """
    if orientation_degrees is None:
        return True
    if abs(wall.azimuth - orientation_degrees) < 45:
        return True
    return False
```

Appr 1. add tag / filter to idfobj / methods.
General, useful later.

built-in

do something here to replace IDF with [ ]

if construction is given, use it. else. use subsurf. construction.
not None = True

replace existing subsurfaces either way. (force or not).

LOC 196. wall: EPBunch.

LOC 163

LOC 113

patches.py → surfaces.py : set_coords (surface : EPBunch, coords : geom related, ggr).
Polygon3D

```
189
190
191 def _is_window(subsurface):
192     if subsurface.key.lower() in {"window", "fenestrationsurface:detailed"}:
193         return True
194
195
196 def window_vertices_given_wall(wall, wwr):
197     # type: (EpBunch, float) -> Polygon3D
198     """Calculate window vertices given wall vertices and glazing ratio.
199
200     :: For each axis:
201         1) Translate the axis points so that they are centred around zero
202         2) Either:
203             a) Multiply the z dimension by the glazing ratio to shrink it vertically
204             b) Multiply the x or y dimension by 0.995 to keep inside the surface
205         3) Translate the axis points back to their original positions
206
207     :param wall: The wall to add a window on. We expect each wall to have four vertices.
208     :param wwr: Window to wall ratio.
209
210     :returns: Window vertices bounding a vertical strip midway up the surface.
211
212     """
213     vertices = wall.coords
214     average_x = sum([x for x, _y, _z in vertices]) / len(vertices)
215     average_y = sum([y for _x, y, _z in vertices]) / len(vertices)
216     average_z = sum([z for _x, _y, z in vertices]) / len(vertices)
217     # move windows in 0.5% from the edges so they can be drawn in SketchUp
218     window_points = [
219         [
220             ((x - average_x) * 0.999) + average_x,
221             ((y - average_y) * 0.999) + average_y,
222             ((z - average_z) * wwr) + average_z,
223         ]
224         for x, y, z in vertices
225     ]
226
227     return Polygon3D(window_points)
228
229
230 def translate_to_origin(idf):
231     # type: (IDF) -> None
232     """Move an IDF close to the origin so that it can be viewed in SketchUp.
233
234     :param idf: The IDF to edit.
235
236     """
237     surfaces = idf.getsurfaces()
238     subsurfaces = idf.getsubsurfaces()
239     shading_surfaces = idf.getshadingsurfaces()
240
241     min_x = min(min(Polygon3D(s.coords).xs) for s in surfaces)
242     min_y = min(min(Polygon3D(s.coords).ys) for s in surfaces)
243
244     translate(surfaces, (-min_x, -min_y))
245     translate(subsurfaces, (-min_x, -min_y))
246     translate(shading_surfaces, (-min_x, -min_y))
247
248
249 def translate(
250     surfaces, vector
251 ):  # type: (Idf_MSequence, Union[Tuple[float, float], Vector2D, Vector3D]) -> None
252     """Translate all surfaces by a vector.
253
254     :param surfaces: A list of EpBunch objects.
255     :param vector: Representation of a vector to translate by.
256
257     """
258     vector = Vector3D(*vector)
259     for s in surfaces:
260         if not s.coords:
261             warnings.warn(
262                 "%s was not affected by this operation since it does not define vertices."
263                 % s.Name
264             )
265             continue
266         new_coords = translate_coords(s.coords, vector)
267         s.setcoords(new_coords)
268
269
270 def translate_coords(coords, vector):
271     # type: (Union[List[Tuple[float, float, float]], Polygon3D], Union[List[float], Vector3D]) -> List[Union[Vector2D, Vector3D]]
272     """Translate a set of coords by a direction vector.
273
274     :param coords: A list of points.
275     :param vector: Representation of a vector to translate by.
276     :returns: List of translated vectors.
277
278     """
279     return [Vector3D(*v) + vector for v in coords]
280
281
282 def scale(surfaces, factor, axes):
```

```
283     # type: (Idf_MSequence, float, str) -> None
284     """Scale all surfaces by a factor.
285
286     :param surfaces: A list of EpBunch objects.
287     :param factor: Factor to scale the surfaces by.
288     :param axes: Axes to scale on.
289
290     """
291     for s in surfaces:
292         if not s.coords:
293             warnings.warn(
294                 "%s was not affected by this operation since it does not define vertices."
295                 % s.Name
296             )
297             continue
298         new_coords = scale_coords(s.coords, factor, axes)
299         s.setcoords(new_coords)
300
301
302 def scale_coords(coords, factor, axes="xy"):
303     # type: (Union[list[Tuple[float, float, float]], Polygon3D], float, str) -> Polygon3D
304     """Scale a set of coords by a factor.
305
306     :param coords: A list of points.
307     :param factor: Factor to scale the surfaces by.
308     :param axes: Axes to scale on.
309     :returns: A scaled polygon.
310
311     """
312     coords = Polygon3D(coords)
313     vertices = []
314     for coord in coords:
315         x = coord[0] * factor if "x" in axes else coord[0]
316         y = coord[1] * factor if "y" in axes else coord[1]
317         z = coord[2] * factor if "z" in axes else coord[2]
318         vertices.append(Vector3D(x, y, z))
319     return Polygon3D(vertices)
320
321
322 def rotate(surfaces, angle):
323     # type: (Union[List[EpBunch], Idf_MSequence], float) -> None
324     """Rotate all surfaces by an angle.
325
326     :param surfaces: A list of EpBunch objects or a mutable sequence.
327     :param angle: An angle in degrees.
328
329     """
330     radians = np.deg2rad(angle)
331     for s in surfaces:
332         if not s.coords:
333             warnings.warn(
334                 "%s was not affected by this operation since it does not define vertices."
335                 % s.Name
336             )
337             continue
338         new_coords = rotate_coords(s.coords, radians)
339         s.setcoords(new_coords)
340
341
342 def rotate_coords(coords, radians):
343     """Rotate a set of coords by an angle in radians.
344
345     :param coords: A list of points.
346     :param radians: The angle to rotate by.
347     :returns: List of Vector3D objects.
348
349     """
350     poly = Polygon3D(coords)
351     rotation = Transformation()._rotation(Vector3D(0, 0, 1), radians)
352     coords = rotation * poly
353     return coords
354
```