IfcOpenShell code examples

From Wiki.OSArch

This page is part of a series about Starting to code. All articles in the series can be found in the Category:Start_coding

IfcOpenShell is a C++ and Python open source software library that helps users and software developers to work with the IFC file format. You can see all articles related articles in the Category:IfcOpenShell.

Before getting started, you may be interested in Using the Python console with BlenderBIM Add-on, to quickly install all the software and development environment you need for writing and running code, without any administrator privileges required. You may also find it useful watching this video (https://www.youtube.com/watch?v=WZPNaAM9ZuQ) which is complementary.

Contents

Crash course

To load a file, you'll need to import IfcOpenShell and store the IFC file in a variable. We'll use the variable `ifc`. The `ifc` will be then used throughout.

```
import ifcopenshell
ifc = ifcopenshell.open('/path/to/your/file.ifc')
```

Let's see what IFC schema we are using:

```
print(ifc.schema) # May return IFC2X3 or IFC4
```

Let's get the first piece of data in our IFC file:

```
print(ifc.by_id(1))
```

But getting data from beginning to end isn't too meaningful to humans. What if we knew a `GlobalId` value instead?

```
print(ifc.by_guid('0EI0MSHbX9gg8Fxwar7lL8'))
```

If we're not looking specifically for a single element, perhaps let's see how many walls are in our file, and count them:

```
walls = ifc.by_type('IfcWall')
print(len(walls))
```

Once we have an element, we can see what IFC class it is:

```
wall = ifc.by_type('IfcWall')[0]
print(wall.is_a()) # Returns 'IfcWall'
```

You can also test if it is a certain class, as well as check for parent classes too:

```
print(wall.is_a('IfcWall')) # Returns True
print(wall.is_a('IfcElement')) # Returns True
print(wall.is_a('IfcWindow')) # Returns False
```

Let's quickly check the STEP ID of our element:

```
print(wall.id())
```

Let's get some attributes of an element. IFC attributes have a particular order. We can access it just like a list, so let's get the first and third attribute:

```
print(wall[0]) # The first attribute is the GlobalId
print(wall[2]) # The third attribute is the Name
```

Knowing the order of attributes is boring and technical. We can access them by name too:

```
print(wall.GlobalId)
print(wall.Name)
```

Getting attributes one by one is tedious. Let's grab them all:

```
print(wall.get_info()) # Gives us a dictionary of attributes, such as {'id': 8, 'type': 'IfcWall', 'GlobalId':
'2_qMTAIHrEYu0vYcqK8cBX', ... }
```

Let's see all the properties and quantities associated with this wall:

```
import ifcopenshell.util
import ifcopenshell.util.element
print(ifcopenshell.util.element.get_psets(wall))
```

Some attributes are special, called "inverse attributes". They happen when another element is referencing our element. They can reference it for many reasons, like to define a relationship, such as if they create a void in our wall, join our wall, or define a quantity take-off value for our wall, among others. Just treat them like regular attributes:

```
print(wall.IsDefinedBy)
```

Perhaps we want to see all elements which are referencing our wall?

```
print(ifc.get_inverse(wall))
```

Let's do the opposite, let's see all the elements which our wall references instead:

```
print(ifc.traverse(wall))
print(ifc.traverse(wall, max_levels=1)) # Or, let's just go down one level deep
```

If you want to modify data, just assign it to the relevant attribute:

```
wall.Name = 'My new wall name'
```

You can also generate new `GlobalId`s:

```
wall.GlobalId = ifcopenshell.guid.new()
```

After modifying some IFC data, you can save it to a new IFC-SPF file:

```
ifc.write('/path/to/a/new.ifc')
```

You can generate a new IFC from scratch too, instead of reading an existing one:

```
ifc = ifcopenshell.file()
# Or if you want a particular schema:
ifc = ifcopenshell.file(schema='IFC4')
```

You can create new IFC elements, and add it either to an existing or newly created IFC file object:

```
new_wall = ifc.createIfcWall() # Will return #1=IfcWall($,$,$,$,$,$,$,$,$) - notice all of the attributes are
blank!
print(ifc.by_type('IfcWall')) # Will return a list with our wall in it: [#1=IfcWall($,$,$,$,$,$,$,$,$)]
```

Alternatively, you can also use this way to create new elements:

```
ifc.create_entity('IfcWall')
```

Specifying more arguments lets you fill in attributes while creating the element instead of assigning them separately. You specify them in the order of the attributes.

```
ifc.create_entity('IfcWall', ifcopenshell.guid.new()) # Gives us
#1=IfcWall('0EI0MSHbX9gg8Fxwar7lL8',$,$,$,$,$,$,$,$)
```

Again, knowing the order of attributes is difficult, so you can use keyword arguments instead:

```
ifc.create_entity('IfcWall', GlobalId=ifcopenshell.guid.new(), Name='Wall Name') # Gives us
#1=IfcWall('0EI0MSHbX9gg8Fxwar7lL8',$,'Wall Name',$,$,$,$,$,$)
```

Sometimes, it's easier to expand a dictionary:

```
data = {
    'GlobalId': ifcopenshell.guid.new(),
    'Name': 'Wall Name'
}
ifc.create_entity('IfcWall', **data)
```

Some attributes of an element aren't just text, they may be a reference to another element. Easy:

```
wall = ifc.createIfcWall()
wall.OwnerHistory = ifc.createIfcOwnerHistory()
```

What if we already have an element from one IFC file and want to add it to another?

```
wall = ifc.by_type('IfcWall')[0]
new_ifc = ifcopenshell.file()
new_ifc.add(wall)
```

Fed up with an object? Let's delete it:

```
ifc.remove(wall)
```

Geometry processing

The usage of IfcOpenShell for geometry processing is currently considered to be moderate to advanced. There are two approaches to processing geometry. One approach is to traverse the `Representation` attribute of the IFC element, and parse it yourself. This requires an in-depth understanding of IFC geometric representations, as well as its many caveats with units and transformations, but can be very simple to extract specific types of geometry. The second approach is to use IfcOpenShell's shape processing features, which will convert almost all IFC representations into a triangulated mesh. Regardless of the source format, once it is in a mesh representation, you may use standard mesh geometry processing algorithms to analyse the geometry. This makes it easier to write generic code for any representation, but may be harder to extract certain geometric features.

The simplest way to get started is to use the `create_shape` function to convert an element into vertices, edges, and faces.

```
import ifcopenshell
import ifcopenshell.geom

ifc_file = ifcopenshell.open('/path/to/your/file.ifc')

element = ifc_file.by_type('IfcWall')[0]

settings = ifcopenshell.geom.settings()
shape = ifcopenshell.geom.create_shape(settings, element)
faces = shape.geometry.faces # Indices of vertices per triangle face e.g. [f1v1, f1v2, f1v3, f2v1, f2v2, f2v3, ...]
verts = shape.geometry.verts # X Y Z of vertices in flattened list e.g. [v1x, v1y, v1z, v2x, v2y, v2z, ...]
materials = shape.geometry.materials # Material names and colour style information that are relevant to this shape
material_ids = shape.geometry.material_ids # Indices of material applied per triangle face e.g. [f1m, f2m, ...]

# Since the lists are flattened, you may prefer to group them per face like so depending on your geometry kernel
grouped_verts = [[verts[i], verts[i + 1], verts[i + 2]] for i in range(0, len(verts), 3)]
grouped_faces = [[faces[i], faces[i + 1], faces[i + 2]] for i in range(0, len(faces), 3)]
```

Once you have a list of vertices, edges, and faces, you can perform any standard mesh algorithm to do more geometric analysis. There are a series of settings you can apply when creating a shape. These are documented here (https://github.com/IfcOpenShell/IfcOpenShell/blob/v0.7.0/docs/geom/settings.md).

All shapes are given an ID to uniquely identify it. These IDs following a naming scheme (https://github.com/IfcOpenShell/IfcOpenShell/issues/866).

If you have a lot of geometry to process, it is advised to use the geometry iterator. This will process shapes using more than one CPU and is significantly faster.

```
import multiprocessing
import ifcopenshell
import ifcopenshell.geom

try:
    ifc_file = ifcopenshell.open('/path/to/your/file.ifc')
except:
    print(ifcopenshell.get_log())
else:
    settings = ifcopenshell.geom.settings()
    iterator = ifcopenshell.geom.iterator(settings, ifc_file, multiprocessing.cpu_count())
    if iterator.initialize():
        while True:
            shape = iterator.get()
            element = ifc_file.by_guid(shape.guid)
            faces = shape.geometry.faces # Indices of vertices per triangle face e.g. [f1v1, f1v2, f1v3, f2v1,
f2v2, f2v3, ...]
            verts = shape.geometry.verts # X Y Z of vertices in flattened list e.g. [v1x, v1y, v1z, v2x, v2y, v2z,
...]
            materials = shape.geometry.materials # Material names and colour style information that are relevant to
this shape
            material_ids = shape.geometry.material_ids # Indices of material applied per triangle face e.g. [f1m,
f2m, ...]

            # Since the lists are flattened, you may prefer to group them per face like so depending on your
geometry kernel
            grouped_verts = [[verts[i], verts[i + 1], verts[i + 2]] for i in range(0, len(verts), 3)]
            grouped_faces = [[faces[i], faces[i + 1], faces[i + 2]] for i in range(0, len(faces), 3)]
            if not iterator.next():
                break
```

IFC Query Syntax

Sometimes, you'd like to query an IFC file for a series of elements. It is possible to write this out manually, by means of functions like `by_type()` and `if` statements, and `for` loops. However, a shorthand query syntax has been invented that makes this process much easier. This is made available in the `ifcopenshell.util.selector` module.

In this simple example below, we use the `#` and `.` prefix to our query tells it that we want to search by IFC `GlobalId` and class type respectively. Whitespace is optional in queries, but is used in these examples for readability.

```
import ifcopenshell.util
from ifcopenshell.util.selector import Selector

ifc = ifcopenshell.open('/path/to/your/file.ifc')
selector = Selector()
element = selector.parse(ifc, '#2MLFd4X2f0jRq28Dvww1Vm') # Equivalent to ifc.by_guid('2MLFd4X2f0jRq28Dvww1Vm')
walls = selector.parse(ifc, '.IfcWall') # This is equivalent to ifc.by_type('IfcWall')
```

It is also possible to search by any attribute:

```
# This is equivalent to searching by GlobalId
elements = selector.parse(ifc, '.IfcSlab[GlobalId = "2MLFd4X2f0jRq28Dvww1Vm"]')

# This finds all slabs with "Precast" (case-sensitive) in the Name
elements = selector.parse(ifc, '.IfcSlab[Name *= "Precast"]')

# This finds all slabs which have a quantified net volume greater than 10 units
elements = selector.parse(ifc, '.IfcSlab[Qto_SlabBaseQuantities.NetVolume > "10"]')

# This finds all walls which have a particular fire rating property
elements = selector.parse(ifc, '.IfcWall[Pset_WallCommon.FireRating = "2HR"]')
```

You can also search by spatial containment using the `@` symbol, for example:

```
# If 0ehnsYoIDA7wC8yu69IDjv is the GlobalId of an IfcBuildingStorey, this gets all of the elements in that storey.
elements = selector.parse(ifc, '@ #0ehnsYoIDA7wC8yu69IDjv')
```

Or by type using the `*` symbol:

```
# If 1uVUwUxTX9Jg1NHVw5KZhI is the GlobalId of an IfcTypeElement, this gets all of the elements of that type.
elements = selector.parse(ifc, '* #1uVUwUxTX9Jg1NHVw5KZhI')
```

You can use `AND` and `OR` statements, using the `&` and `|` symbols respectively:

```
# This gets all the 2HR fire rated walls in a particular building storey:
elements = selector.parse(ifc, '@ #0ehnsYoIDA7wC8yu69IDjv & .IfcWall[Pset_WallCommon.FireRating = "2HR"]')

# This is equivalent to walls = ifc.by_type('IfcWall') + ifc.by_type('IfcSlab'), i.e. all walls and slabs
elements = selector.parse(ifc, '.IfcWall | .IfcSlab')
```

You can also use parenthesis to group queries and combine them:

```
# This gets all walls and slabs in a particular building storey
elements = selector.parse(ifc, '@ #0ehnsYoIDA7wC8yu69IDjv & ( .IfcWall | .IfcSlab )')
```

```
# This gets all walls and slabs in a particular space (if IfcRelSpaceBoundary exist)
elements = selector.parse(ifc, '@@ .IfcSpace & ( .IfcWall | .IfcSlab )')
```

Exploring IFC schema

Get Schema :

```
import ifcopenshell
schema = ifcopenshell.ifcopenshell_wrapper.schema_by_name("IFC4")
```

Explore class attributes :

```
>>> ifc_pipe = schema.declaration_by_name("IfcPipeSegment")
>>> ifc_pipe.all_attributes()
(<attribute GlobalId: <type IfcGloballyUniqueId: <string>>>,
 <attribute OwnerHistory?: <entity IfcOwnerHistory>>,
 <attribute Name?: <type IfcLabel: <string>>>,
 <attribute Description?: <type IfcText: <string>>>,
 <attribute ObjectType?: <type IfcLabel: <string>>>,
 <attribute ObjectPlacement?: <entity IfcObjectPlacement>>,
 <attribute Representation?: <entity IfcProductRepresentation>>,
 <attribute Tag?: <type IfcIdentifier: <string>>>,
 <attribute PredefinedType?: <enumeration IfcPipeSegmentTypeEnum: (CULVERT, FLEXIBLESEGMENT, GUTTER, NOTDEFINED,
RIGIDSEGMENT, SPOOL, USERDEFINED)>>)

>>> ifc_pipe.attribute_count()
9

>>> ifc_pipe.attribute_by_index(0)
<attribute GlobalId: <type IfcGloballyUniqueId: <string>>>

>>> ifc_pipe.attribute_index("Description")
3

>>> predefined_type = ifc_pipe.attribute_by_index(8)
>>> predefined_type.name()
'PredefinedType'

>>> predefined_type.optional()
True

>>> predefined_type.type_of_attribute()
<enumeration IfcPipeSegmentTypeEnum: (CULVERT, FLEXIBLESEGMENT, GUTTER, NOTDEFINED, RIGIDSEGMENT, SPOOL,
USERDEFINED)>

# from name you can then recursively explore how create an entity as described above
>>> predefined_type.type_of_attribute().declared_type().name()
'IfcPipeSegmentTypeEnum'
```

Get supertype :

```
>>> ifc_pipe.supertype()
<entity IfcFlowSegment>
```

Get subtypes :

```
>>> super_type = ifc_pipe.supertype()
>>> super_type.subtypes()
(<entity IfcCableCarrierSegment>,
 <entity IfcCableSegment>,
 <entity IfcDuctSegment>,
 <entity IfcPipeSegment>)
```

Property and quantity sets (Pset and Qto)

First import util for psets and qtos and initialise main class with schema name:

```
>>> import ifcopenshell.util.pset
>>> from ifcopenshell import util
>>> pset_qto = util.pset.PsetQto("IFC4")
```

Get all applicable psets/qtos names for a certain class :

```
>>> pset_qto.get_applicable_names("IfcMaterial")
['Pset_MaterialCombustion',
 'Pset_MaterialCommon',
 'Pset_MaterialConcrete',
 'Pset_MaterialEnergy',
 'Pset_MaterialFuel',
 'Pset_MaterialHygroscopic',
 'Pset_MaterialMechanical',
 'Pset_MaterialOptical',
 'Pset_MaterialSteel',
 'Pset_MaterialThermal',
 'Pset_MaterialWater',
 'Pset_MaterialWood',
 'Pset_MaterialWoodBasedBeam',
 'Pset_MaterialWoodBasedPanel']
```

You can also get a list of all applicable pset/qto templates to get more information:

```
>>> pset_qto.get_applicable("IfcMaterial")[0]
#22772=IfcPropertySetTemplate('3Eppg0qUmHu000025QrE$V',$,'Pset_MaterialCommon','A set of general material
properties.',.PSET_TYPEDRIVENOVERRIDE.,'IfcMaterial',(#22775,#22778,#22781))
```

You can also retrieve a standard Pset by name:

```
>>> pset_template = pset_qto.get_by_name("Pset_MaterialCommon")
>>> pset_template
#22772=IfcPropertySetTemplate('3Eppg0qUmHu000025QrE$V',$,'Pset_MaterialCommon','A set of general
material,'IfcMaterial',(#22775,#22778,#22781))
>>> pset_template.get_info()
{'id': 22772,
 'type': 'IfcPropertySetTemplate',
 'GlobalId': '3Eppg0qUmHu000025QrE$V',
 'OwnerHistory': None,
 'Name': 'Pset_MaterialCommon',
 'Description': 'A set of general material properties.',
 'TemplateType': 'PSET_TYPEDRIVENOVERRIDE',
 'ApplicableEntity': 'IfcMaterial',
 'HasPropertyTemplates': (#22775=IfcSimplePropertyTemplate('3I0i60qUmHu000025QrE$V',$,'MolecularWeight','Molecular
weight of material (typically gas).',.P_SINGLEVALUE.,'IfcMolecularWeightMeasure','',$,$,$,$,.READWRITE.),
  #22778=IfcSimplePropertyTemplate('30MAA0qUmHu000025QrE$V',$,'Porosity','The void fraction of the total volume
occupied by material (Vbr - Vnet)/Vbr.',.P_SINGLEVALUE.,'IfcNormalisedRatioMeasure','',$,$,$,$,.READWRITE.),
  #22781=IfcSimplePropertyTemplate('3TjUq0qUmHu000025QrE$V',$,'MassDensity','Material mass
density.',.P_SINGLEVALUE.,'IfcMassDensityMeasure','',$,$,$,$,.READWRITE.))}
```

File validation

Ifcopenshell allows you to validate a file against its corresponding schema.

From command line

```
python -m ifcopenshell.validate some_file.ifc
```

Inside a script

It looks similar to command line way but you need to provide a logger (standard logger or json_logger as below).

```
import ifcopenshell
import ifcopenshell.validate

ifc_file = ifcopenshell.open("some_file.ifc")
json_logger = ifcopenshell.validate.json_logger()
ifcopenshell.validate.validate(ifc_file, json_logger)
json_logger.statements
```

Date and time in IFC

Date and time conversion can be error prone. Fortunately IFC has chosen wide used definitions so standard python datetime (https://docs.python.org/3/library/datetime.html) and time (https://docs.python.org/3/library/time.html) modules have tools to handle most of them.

IfcTimeStamp

IfcTimeStamp (https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/link/ifctimestamp.htm) is the elapsed number of seconds since 1 January 1970, 00:00:00 UTC. See wikipedia (https://en.wikipedia.org/wiki/Unix_time) for more detailed explanations. It is typically used to record when you create or modify an IFC object. Warning while python output a float, IFC expect an integer.

```
import time
# To store current time
>>>int(time.time())
1610829915
# To convert it to datetime (UTC+1 in this example)
>>> import datetime
>>> datetime.date.fromtimestamp(0)
datetime.date(1970, 1, 1, 1, 0)
>>> datetime.date.fromtimestamp(1610829915)
datetime.date(2021, 1, 16, 21, 45, 15)
```

IfcDateTime

IfcDateTime (https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/link/ifcdatetime.htm) is stored in ISO 8601 format : YYYY-MM-DDThh:mm:ss Convert back and forth :

```
from datetime import datetime
>>> dt = datetime.fromisoformat("2021-01-16T21:45:15")
>>> dt
datetime.datetime(2021, 1, 16, 21, 45, 15)
>>> dt.isoformat()
'2021-01-16T21:45:15'
```

IfcDate

IfcDate (https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/link/ifcdate.htm) is similar to IfcDateTime but without the time part : YYYY-MM-DD You can use `datetime.date` instead of `datetime.datetime` the same way.

IfcTime

IfcTime (https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/link/ifctime.htm) is similar to IfcDateTime but without the date part : hh:mm:ss You can use `datetime.time` instead of `datetime.datetime` the same way.

IfcDuration

IfcDuration (https://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/link/ifcduration.htm) is used to store a time interval (eg. time to perform a task) in ISO 8601 format. There is apparently no ready to use standard python module to handle it. It can be handled using regular expressions. For format given as example in IFC documentation :

```
>>> import re
>>> re.match("P(\d+)Y(\d+)M(\d+)DT(\d+)H(\d+)M(\d+)S", "P2Y10M15DT10H30M20S").groups()
('2', '10', '15', '10', '30', '20')
```

To handle all available formatting options defined in ISO 8601 (https://en.wikipedia.org/wiki/ISO_8601#Durations) a more complex regular expression is required like following. You can test it on pythex (https://pythex.org/?regex=P(%3F%3A(%3FP%3Cyear%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)%5BY-%5D)%3F(%3F%3A(%3FP%3Cmonth%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)%5BM-%5D)%3F(%3F%3A(%3FP%3Cweek%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)W)%3F(%3F%3A(%3FP%3Cday%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)D%3F)%3F(%3F%3AT(%3F%3A(%3FP%3Chour%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)%5BH%3A%5D)%3F(%3F%3A(%3FP%3Cminute%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)%5BM%3A%5D)%3F(%3F%3A(%3FP%3Csecond%3E%5Cd%2B%3F%5B%2C%5C.%5D%3F%5Cd*)S%3F)%3F)%3F&test_string=P3Y6M4DT12H30M5S%0AP1M%0APT1M%0AP1DT12H%0AP0%2C5Y%0AP0.5Y%0APT36H%0AP1DT12H%0AP0003-06-04T12%3A30%3A05%0AP05Y%0AP0..5Y%0AP0%2C.5Y&ignorecase=0&multiline=0&dotall=0&verbose=0).

```
P(?:(?P<year>\d+?[,\.]?\d*)[Y-])?(?:(?P<month>\d+?[,\.]?\d*)[M-])?(?:(?P<week>\d+?[,\.]?\d*)W)?(?:(?
P<day>\d+?[,\.]?\d*)D?)?(?:T(?:(?P<hour>\d+?[,\.]?\d*)[H:])?(?:(?P<minute>\d+?[,\.]?\d*)[M:])?(?:(?
P<second>\d+?[,\.]?\d*)S?)?)?
```

IfcCalendarDate

Warning: Deprecated: IFC2X3 only

Similar to `IfcDate` but in a different format. It has 3 attribute for day, month and year.

```
>>> from datetime import date
>>> import ifcopenshell
>>> ifc = ifcopenshell.file(schema="IFC2X3")
>>> ifc_date = ifc.createIfcCalendarDate(16, 1, 2021)
>>> py_date = date(ifc_date.YearComponent, ifc_date.MonthComponent, ifc_date.DayComponent)
>>> py_date
datetime.date(2021, 1, 16)
>>> ifc.createIfcCalendarDate(py_date.day, py_date.month, py_date.year)
#4=IfcCalendarDate(16,1,2021)
```

Samples from web

Subject	Language	Version	Additional infos
Optimize IFC files (https://academy.ifcopenshell.org/posts/ifcopenshell-optimizer-tutorial/)	python	0.6.0-0d93633	
Calculate Differences of IFC files with Hashing (https://academy.ifcopenshell.org/posts/calculate-differences-of-ifc-files-with-hashing/)	python	0.6.0-0d93633	
Understanding placements in IFC using IfcOpenShell and FreeCAD (https://pythoncvc.net/?p=851)	python	0.6.0a1	FreeCAD 0.18 git
Using IfcOpenShell to parse IFC files with Python (https://thinkmoult.com/using-ifcopenshell-parse-ifc-files-python.html)	python		
Read geometry as Boundary Representation in FreeCAD (https://pythoncvc.net/?p=839)	python	0.6.0a1	FreeCAD 0.18 git
Read IFC geometry as triangle meshes in FreeCAD (https://pythoncvc.net/?p=822)	python	0.6.0a1	FreeCAD 0.18 git
Using IfcOpenShell and C++ to generate Alignments through the IFC 4×1 schema (https://academy.ifcopenshell.org/posts/using-ifcopenshell-and-c%2B%2B-to-generate-alignments-through-the-ifc-4x1-schema/)	python/C++		
Creating a simple wall with property set and quantity information (https://academy.ifcopenshell.org/posts/creating-a-simple-wall-with-property-set-and-quantity-information/)	python		
Using IfcOpenshell and pythonOCC to generate cross sections directly from an IFC file (https://academy.ifcopenshell.org/posts/using-ifcopenshell-and-pythonocc-to-generate-cross-sections-directly-from-an-ifc-file/)	python	<= 0.6 ?	pythonOCC 0.16.0 ?
Using the parsing functionality of IfcOpenShell interactively (https://academy.ifcopenshell.org/posts/using-the-parsing-functionality-of-ifcopenshell-interactively/)	python		
Using IfcOpenShell and pythonOCC to construct new geometry (https://academy.ifcopenshell.org/posts/using-ifcopenshell-and-pythonocc-to-construct-new-geometry/)	python	<= 0.6	pythonOCC 0.16.0
Various ifc creation examples (little house, geometry etc...) (https://github.com/IfcOpenShell/IfcOpenShell/tree/v0.6.0/src/examples)	C++		

Retrieved from "https://wiki.osarch.org/index.php?title=IfcOpenShell_code_examples&oldid=4648"